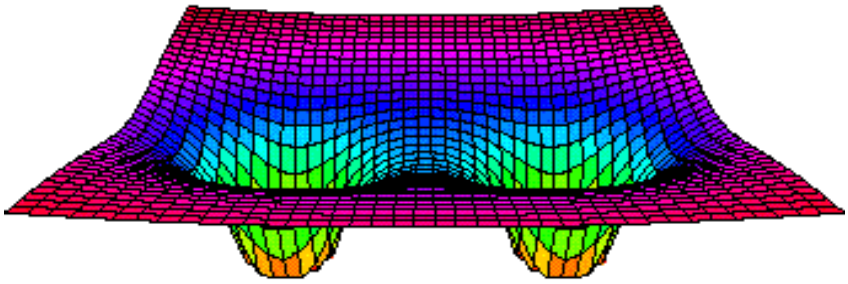
CHAPTER 1

Introduction



Computer languages for scientific and engineering applications have evolved in many directions since the early days of FORTRAN. Many features such as structured, procedural and object oriented programming have worked their ways into the modern repertoire of computer languages. Still, these languages lack a shared notation with the analytical discussion of many fields, especially where tensors are concerned. GRPP is a high level computer language for developing scientific and engineering software that is fundamentally represented by tensors. It serves as a front end to the C language, translating GRPP tensor notation into ANSI C code.

Black holes and gravity waves

The development of GRPP precipitated out of the needs of a project to study the coalescence of a binary system of black holes at the University of Florida. Such a system is described by Einstein's general theory of relativity. It is a theory rich with tensors used to represent the macroscopic properties of the universe we live in. The binary black hole system itself is very important as a source for strong gravitational radiation. As the black holes orbit about each other they slowly spiral in by emitting energy in the form of gravity waves. Detectors for observing these gravity waves

are planned by the US and European governments. They will have a very good chance to observe such events so having a sound theoretical understanding of these systems will aid in their insightful detection.

Developing software to model the binary system of black holes required extensive code development and testing using the conventional C language. Months of programming time went into developing each model. Tensor representation and tensor analysis with C was a principle source of the work load in developing the model. To circumvent this time sink, I found another - lex and yacc.

After a few of months in front of the terminal I had the first version of GRPP in hand. It came complete with a grammar for the tensor analysis needed to write very compact source code for most midsize tensor analysis projects. But in order to move on to larger projects, tensors needed to be passed to and from functions. This lead to the development of a second version of GRPP.

GRPP and the C language

The C language is marvelous target for a high level computer language front end. It has a wealth of features based on structured and procedural programming practices. In addition, it is very portable from one computer platform to another, at least as far as computer codes go. It is also becoming the standard for scientific and engineering software development. For these reason, GRPP has been written to take combined GRPP tensor notation along with ANSI standard C code as input from a file and produce purely ANSI C code in an output source and header file. This also gives the GRPP developer the flexibility to use existing software and utilities in concert with GRPP. The source code for GRPP itself is written in a combination of lex, yacc and C code, making GRPP available for most computer architectures. In addition, in the event that you have access to GRPP on one platform and a C compiler on a second platform, the output of GRPP can simply be file copied over to the second computer having the C compiler.

Mixing of GRPP tensor code and standard C code is handled by the lexical analyzer with GRPP through the use of what is referred to as the GRPP code BLOCK. This is a BLOCK of code separated from any surrounding ANSI C code by matching pairs of ‘$$’ characters. This reduces the burden on the preprocessor and simplifies the syntax of GRPP. It also avoids duplication of support for syntax and error checking that are already provided for by the C language compiler. Finally, support

for other C style languages such as C++ and Objective-C are in a better position to be directly supported by GRPP with this scheme:

```
...
ANSI C code
$$
GRPP code BLOCK
$$
ANSI C code
$$
Another GRPP code BLOCK
$$
More ANSI C code
...
```

GRPP code BLOCK

The preprocessing action of GRPP is only active for the duration of code that lies between matching pairs of '$$' characters.

Using GRPP with C++ and Objective-C

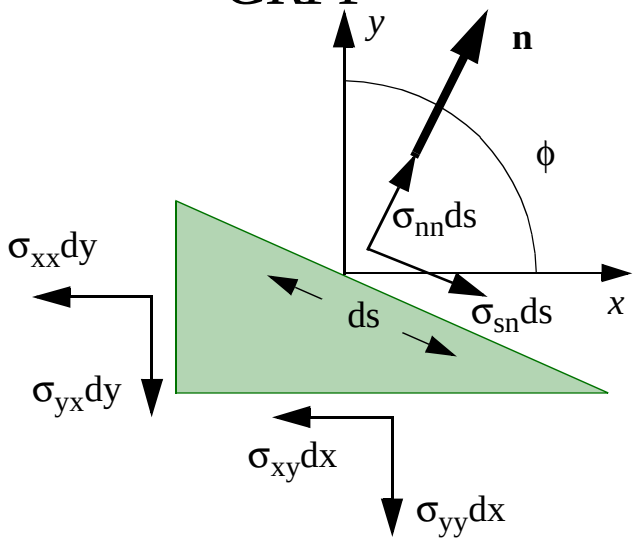
Since both C++ and Objective-C are extensions of the C language, one can with a bit of care, develop code for incorporation into these languages with GRPP. The lexical analyzer for GRPP was developed to navigate through a convoluted mixing of C and GRPP tensor syntax. With the correct use of the GRPP code BLOCK, the preprocessor can navigate successfully through mixed GRPP and C++ or Objective-C code. Simply put, this amounts to replacing the "ANSI C code" blocks in the diagram above with C++ or Objective-C code blocks.

The ability to mix C++ or Objective-C code using the GRPP code BLOCK has not been as thoroughly tested as the mixing of C code with GRPP tensor code has been and little "gotchas" may be lurking about, so be aware and report findings back to the author.

Unix, GRPP and the GRCC script

Currently, GRPP has been implemented on Sun, DEC, NeXT and SGI workstations running versions of Unix. GRPP comes with GRCC, a script program that shortens the development cycle by automatically calling GRPP and passing the output directly on to the Unix cc compiler. This script can be hand tailored to your particular computer setup. For example, replacing the C compiler cc with the GNU gcc compiler.

CHAPTER 2 *Tensors and GRPP*



Tensors are multi-linear mathematical objects which include and extend the concepts of vectors. All of vector analysis is simply a subset of the more general techniques of tensor analysis. Vectors and tensors play an important role in all of the physical sciences such as physics and geology and in engineering and mathematics. They are used to express the general covariance found in the laws of nature. Covariance is a way of describing that the laws of nature are the same for all observers, independent of choice of coordinates and inertial frames of reference. This view of

general covariance is beautifully illustrated in Einstein's general theory of relativity. Einstein's field equations for describing the relationship between matter and space-time curvature are given by

$$G^{ij} = R^{ij} - {}^{1}/_{2}\, g^{ij} R = 8\pi T^{ij}$$

where the indices 'i' and 'j' range over the four dimensions of space-time. Here the indices are written as superscripts indicating that they are contravariant indices. Subscripted indices represent covariant indices. The implied sum over these indices, known as the Einstein summation convention, means that the field equation actually represents 16 equations. Other places where the general covariances of tensor analysis are used include electromagnetism, optics and mechanical stress analysis.

A thoughtful reader should see two potential difficulties with a computer language supporting tensor. First, indices are themselves variables within the tensor. Variables names in the C language do not support indices. Second, neither ASCI text editors nor the C language support superscripts and subscripts.

To overcome the first, GRPP uses a tensor variable name having 3 parts.

- A descriptive name less than 24 characters long, first letter being *a-zA-Z
- A delimiting '$' character immediately after the name
- From one to four indices characters depending on the rank of the tensors

The second difficulty is overcome by using upper case letters to represent contravariant indices and lower case letters to represent covariant indices.

Combining these, the tensor G^{ij} would look like G$IJ in GRPP code and the tensor T^{ij} would look like T$IJ. Example of tensors of ranks one through 4 are

- Rank 1: R_i, S^j have GRPP counterparts R$i and S$J
- Rank 2: R_{ij}, $S^i_{\ j}$, $T_i^{\ j}$, and U^{ij} given by Rij, SIj, T$iJ, and U$IJ
- Rank 3: R_{ijk}, $S^i_{\ jk}$, $T_i^{\ j}{}_k$, $U_{ij}^{\ \ k}$, etc. given by Rijk, SIjk, TiJk, UijK, etc.
- Rank 4: $R^i_{\ jk}{}^l$, $S_i^{\ jk}{}_l$, $T^i_{\ j}{}^k{}_l$, etc. given by R$IjkL, S$iJKl, T$IjKl, etc.
- Tensors above rank 4 are not supported in this version of GRPP.

GRPP translates these tensor into an ANSI C structure. These structures are defined in the header file produced by GRPP. They are ‘typedef’ so as to be available to the software developer. They have the form ‘TENSOR_C’, ‘TENSOR_c’, etc. for GRPP tensor of form ‘R$I’, ‘R$i’, etc. Using these structures will be covered in more detail in the section on passing tensors between functions.

Setting up the coordinate system

A tensor can exist as a geometric object independently of the choice of coordinate system. Application of tensor analysis however, usually require that a choice of coordinates be made. This is sometimes referred to as a coordinate representation of tensors and is the representation used by GRPP. The indices are simply dummy value fields that can take on any of the coordinates found in the orthogonal coordinate system.

Consider three dimensional Cartesian space. This is typically represented by the coordinates (x,y,z) which define the right handed basis for discussions in ordinary space. To set up GRPP to use this coordinate system as the possible values on indices in tensors, begin the GRPP source code with a few lines similar to this

```
$$ /* set up coordinate system for GRPP */
coordinates$(x,y,z);
$$
...
```

GRPP code BLOCK

Notice the use of the comment statement following the opening ‘$$’. Unnested C comment statements are allowed within a GRPP code BLOCK. Using them will make your code easier to follow. Also, the use of the lower case character set in (x,y,z) was not a requirement. The statement ‘coordinates$(X,Y,Z);’ or any combination of lower and upper case characters are allowed. But using lower case is more conventional and easier to read. However, the ‘coordinate$’ function must be in lower case. Once this statement is read by the preprocessor, it will populate internal buffers with the coordinates and the dimension of the space that the tensors describe. The preprocessor also adds comments to the output C code describing these coordinates and the dimension.

Setting up the indices

In a component representation of tensors, indices are used as place holders in the covariant representation of physical laws. GRPP needs to know what indices to use in the internal tensor analysis. In addition, requiring the user to specify the choice of indices allows for a more robust error checking algorithm. To specify the indices to be used by GRPP use a statement similar to

```
$$ /* set up coordinate system for GRPP */
coordinates$(x,y,z);
/* Set up tensor indices */
indices$(i,j,k,l,m);
$$
...
```

GRPP code BLOCK

which indicates the covariant (lower) tensor indices will be the lower case (i,j,k,l,m) and the contravariant (upper) tensor indices will be upper case (I,J,K,L,M). In a few cases GRPP will make an internal choice for a dummy index. This will always be the last index in the indices$ statement. Remember that if you are working with rank 4 tensors you will need at least 5 indices in the ‘indices$ ‘statement. However, you can have more if you like. Case in not important, but they can not be letters already assigned to a coordinate. All indices must consist of a single letter and the ‘indices$’ function must be in lower case characters.

Having defined both the coordinates and the indices, GRPP will construct a C header file containing some comments about GRPP and the original input file’s associated header file. This header file takes its name from the name of the input file’s appended with the string ‘_grpp.h’. The remainder of this header file contains the typedef statements for the tensor structures used by ANSI C to hold the tensor components. There is also a ‘COMPONENT’ macro in the header which is defined to be ‘double’. It holds the data type for the tensor components. By editing the header file and changing the definition of ‘COMPONENT’ to something else like ‘float’, the underlying data type for the tensor components can me modified. The header file also contains some useful macros containing basic information about the coordinates, indices and the dimension.

- GRPP_DIMENSION ... the dimension of the tensor space
- GRPP_INDICES ... the contravariant and covariant indices used
- GRPP_COORDINATES ... the coordinates used

Using the simple ‘GRPP block CODE’ of the previous page results in the following first few lines in the header file

```
/* General Relativity PreProcessor */
/*        Version 2.1            */
/*                               */
/*    Copyright (c) 1992-1999    */
/*    by TensorSoft     */
/*    All Rights Reserved        */
/*                               */
/* GRPP header file: test_grpp.h */


#define GRPP_DIMENSION 3
#define GRPP_INDICES "IiJjKkLlMm"
#define GRPP_COORDINATES "XxYyZz"


#define COMPONENT double

typedef struct {
            COMPONENT x,y,z;
           } TENSOR_c;

typedef struct {
            COMPONENT X,Y,Z;
           } TENSOR_C;

typedef struct {
            COMPONENT xx,xy,xz;
            COMPONENT yx,yy,yz;
            COMPONENT zx,zy,zz;
           } TENSOR_cc;
```

GRPP Header File

Of course this file varies from application to application. Each GRPP source file will have its own associated header file. Care must be taken not to conflict with the typedef tensors between header files. GRPP uses the same naming scheme in all header files regardless of the coordinates and indices so their inclusion should be limited to that file that it is meant for. GRPP automatically adds a line to the output ANSI C source file for including the associated header file so the user generally doesn’t have to be concerned with these details.

Defining vectors and tensors

Tensors must be typed cast in GRPP just like any other data type in a highly type sensitive language. Variables are type cast as GRPP tensors with the following statements

```
$$ /* set up coordinate system for GRPP */
coordinates$(x,y,z);
/* Set up tensor indices */
indices$(i,j,k,l,m);
$$

#include <math.h>
#include <stdio.h>

main()
{
int counter;
double distance;
$$ /* type cast tensors of rank 1 through 4 */
rank1 r$i, *u$I;
rank2 g$ij, g$IJ, Ricci$Ij;
rank3 g$ij,k, g$ij;k, Christoffel$Ijk;
rank4 Riemannian$Ijkl;
$$
...
}
```

GRPP code BLOCK

There are several restrictions that apply when type casting GRPP tensors.

- Use rank1, rank2, rank3 or rank4 as the types
- From 1 to 6 tensors of the same rank can be cast on a single line
- Separate each tensor on the line with a “,<space>” - a single comma fails
- You may use as many rank1 through rank4 type cast statements as you need
- An * as the first character of a tensor name assigns a pointer to the tensor

The GRPP preprocessor will convert these to the necessary C structures found in the header file on output to the C source file. These type casts result in the following lines of code in the output C file

```
/* Open GRPP block */
/* Type cast tensors of rank 1 through 4 */

TENSOR_c r_c ;
TENSOR_C u_C ;

TENSOR_cc g_cc ;
TENSOR_CC g_CC ;
TENSOR_Cc Ricci_Cc ;

TENSOR_ccc _d_g_ccc ;
TENSOR_ccc _D_g_ccc ;
TENSOR_Ccc Christoffel_Ccc ;

TENSOR_Cccc Riemannian_Cccc ;

/* Close GRPP block */
```

GRPP output C code

You have probably noticed that a couple of thing were skimmed over in the type casting. Four tensors were declared with the same tensor prefix name; gij, gIJ, g$ij,k and g$ij;k. Each of them is a distinct structure in the output C code. The first is a second rank tensor name ‘g’ with two covariant indices, the second is a second rank tensor named ‘g’ with two contravariant indices, the third is a third rank tensor named ‘g’ which actually employees the convention of a comma for partial derivative (this is why a “,<space>” is needed as a delimiter), the fourth is a third rank tensor name ‘g’ which employees the convention of a semicolon for a covariant derivative. Each is a distinct tensor and will be treated as such through the preprocessing of the GRPP source code. This enhances the ability to use conventional notations from differential geometry in GRPP source code.

Also notice that the preprocessing of the tensor names has append an underscore followed by a number of ‘c’ or ‘C’ characters corresponding to the rank and whether the index was a superscript ‘C’ or a subscript ‘c’. The upper case ‘C’ represents Contravariant and the lower case ‘c’ represents covariant indices. The ‘$’ character has also been translated to an underscore character indicating a successful translation.

Passing tensors to and from C functions

ANSI C supports passing structures as arguments to functions. Additionally, pointers to structures provides a faster method of passing information within structures and allows the values to be modified by the called function. GRPP has support for passing structures and pointers to structures to functions. Using tensors as arguments is a bit more trick however. This is do to the ‘c/C’ characters that are appended to the tensor name by the preprocessor. An example will illustrate the technique

```
$$ /* set up coordinate system for GRPP */
coordinates$(x,y,z);
/* Set up tensor indices */
indices$(i,j,k,l,m);
$$
#include <math.h>
#include <stdio.h>

main()
{
double x,y,z;
$$ /* type cast tensors */
rank2 g$ij;
$$
/* define point in space (x,y,z) */
x = 1.0; y = 2.0; z = 3.0;
/* assign values to metric at point (x,y,z) */
/* passing tensor as a pointer to structure */
assign_metric(x,y,z,&g_cc);
}

void assign_metric(x,y,z,metric_cc)
double x,y,z;
TENSOR_cc *metric_cc;
{
$$
/* do assignments to “*metric” */
$$
}
```

GRPP code BLOCK

It is very important to append the underscore and ‘c/C’ characters as in the case of ‘metric_cc’. However, as will be made more clear in the next section, within a

GRPP code block the variable name to be used is actually '*metric'. There is a second way to code the 'assign_metric' function. Consider this approach.

```
$$ /* set up coordinate system for GRPP */
coordinates$(x,y,z);
/* Set up tensor indices */
indices$(i,j,k,l,m);
$$
#include <math.h>
#include <stdio.h>

main()
{
double x,y,z;
$$ /* type cast tensors */
rank2 g$ij;
$$
/* define point in space (x,y,z) */
x = 1.0; y = 2.0; z = 3.0;
/* assign values to metric at point (x,y,z) */
/* passing tensor as a pointer to structure */
assign_metric(x,y,z,&g_cc);
}

void assign_metric(x,y,z,metric_cc)
double x,y,z;
$$
rank2 *metric$ij;
$$
{
$$
/* do assignments to "*metric" */
$$
}
```

GRPP code BLOCK

This will have identical results, and you may find the type casting to be more consistent with the general method of defining tensor. But within the argument list, the form 'metric_cc' must still be used.

Passing tensors by value is very similar. Just replace '*metric' with 'metric' as in this next example.

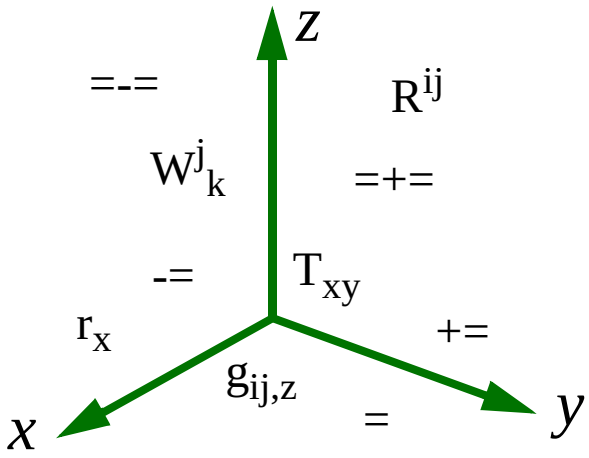
```
$$ /* set up coordinate system for GRPP */
coordinates$(x,y,z);
/* Set up tensor indices */
indices$(i,j,k,l,m);
$$
#define EPS 1.0e-5
#include <math.h>
#include <stdio.h>

main()
{
double x,y,z,dx,dy,dz,ds,distance();
$$ /* type cast tensors */
rank2 g$ij;
$$
x = 1.0; y = 2.0; z = 3.0;
assign_metric(x,y,z,&g_cc);
/* make (dx,dy,dz) small */
dx = dy = dz = EPS;
/* pass a tensor by value */
ds = distance(dx,dy,dz,g_cc);
}

void assign_metric(x,y,z,metric_cc)
double x,y,z;
$$
rank2 *metric$ij;
$$
{
$$
/* do assignments to "*metric" */
$$
}

double distance(dx,dy,dz,metric_cc)
double dx,dy,dz;
$$
rank2 metric$ij;
$$
{
$$
/* do assignments to "metric" */
$$
}
```

GRPP code BLOCK

In all cases, the call to the function must not be in a GRPP code BLOCK.

CHAPTER 3

Tensor Assignment



The full compliment of assignment operators found in C are also available to GRPP source code. These are the '=', the '+=' and the '-=' operators. They make assignments as you would expect from their application in C. In fact, they are simply passed on to the C compiler. GRPP also includes two new assignment operators, '=+=' for making assignments to symmetric tensor components and '=-=' for making assignments to asymmetric tensor components.

Assignment operators

Assignment of Tensors within GRPP can be a bit tricky. As a general rule one should only assign a tensor of rank 'n' to another tensor of rank 'n'. However, GRPP allows assignment of tensors of any rank to a scalar. This is extremely useful for initializing a large number of components with a single statement. However, the cascading '=' of C is not allowed. Add assignments should be limited to one per line. Here are some examples using tensor assignments.

- **g$ij = 0.0**; Assign the value of 0.0 to all component of the tensor **g**
- **A$ijk += sin(k*x);** Increase the values by sin(kx) for all components of **A**
- **B$ij -= 15.7;** Decrease the value of each component of **B** by 15.7

The use of the symmetric and asymmetric assignment operators will be described after discussing the assignment of individual components of a tensor.

Assigning components of tensors

If tensor components were all alike there wouldn't be any need for GRPP. Calculating for one component would give the results for all other components. Since components of tensors have the capability of being unique and independent of other components in the same tensor, a means of assigning individual components is required. Assuming that we are still working with tensors in the (x,y,z) Cartesian space, then here are some typical examples of tensor component assignments.

- **g$xx = 3.3;** Assign the value of 3.3 to the xx component of the tensor **g**
- **r$z -= 1.0;** Decrease the value of the z component of the vector **r** by 1.0
- **g$xy = sin(w*t);** Assign the value of the sin(wt) to the xy component of **g**
- **A$xz += B$zy;** Increase the value of the xz component of the tensor **A** by the zy component of tensor **B**

Each component is simply a scalar and each assignment is simply a scalar assignment. Components are that simple.

Assigning tensors as objects

It is possible to create somewhat more complex assignment statements using the entire tensor or even sub-ranges of tensors. As discussed earlier, assigning tensors of equal rank to one and other is a necessary and a logical function for tensor assignment operators. Additionally, assigning a sub-range of components within a tensor may be useful at times. Here are some examples of these operations.

- **R$Ijk = T$Ijk;** Assign each component of the rank 3 tensor **R** to each component of another rank 3 tensor **T**
- **W$Ijj = U$Ij;** Assign a sub-range of a rank 3 tensor **W** to a rank 2 tensor **U**

- **g$ii = diagonal$i;** Assign the diagonal components of a rank 2 tensor to a vector
- **U$Ixz -= 63.0;** Decrease the value of the Xxz, Yxz, and Zxz components of the tensor **U** by 63.0
- **W$Izz = V$Iy;** Assign values to the Xzz, Yzz and Zzz components of the tensor **W** with the values of the Xy, Yy and Zy components of the tensor **V**
- **g$ij,x = diff$ij;** Assign the components of the tensor **diff** to the partial derivative with respect to x of the tensor **g**
- **g$xx,k = grad$k;** Assign the components of the tensor **grad** to the gradient of the xx component of the **g** tensor

Once you become familiar with these operation, it will become clear that each assignment is unambiguous. This should provide enough insight into the flexibility for getting the general idea of tensor component assignment.

Asymmetric and symmetric assignments

Many tensors have the property that upon exchange of indices, the values of tensor components are equal (symmetric) or negatives of each other (asymmetric). GRPP has two operators for assigning components of asymmetric and symmetric tensors. For symmetric assignment the '=+=' operator is used, and for asymmetric assignment the '=-=' operator is used. Here are some examples.

- **G$xy =+= 10;** same as **G$xy = 10; G$yx = 10;** (assigns 2 components)
- **R$Ixjy =+= 5;** same as **R$Ixjy = 5; R$Iyjx = 5;** (assigns 18 components)
- **A$xzK =-= 3;** same as **A$xzK = 3; A$zxK = -3;** (assigns 6 components)
- **B$Ixz =-= T$I;** same as **B$Ixz = T$I; B$Izx = -T$I;** (assigns 6 components)

In these examples, as before, the (x,y,z) Cartesian coordinates are being used. There are two things that must be true in order to use these operators. The tensor being assigned must have

- exactly two coordinates specified
- these two coordinates must be different

After satisfying these requirement, the tensor must use indices to fill out its rank. Using these operators on tensor of any other form can be dangerous and is not supported.

Using arrays with tensor assignments

Data contained in tensor components can easily be exchanged with data contained in C array elements using the array syntax of GRPP. To assign the elements of an array to the components of a tensor use the following syntax

- **A$i = array_name[];** Assign 1D array to a vector
- **B$Ij = array_name[];** Assign 2D array to a rank 2 tensor
- **C$ijK = array_name[];** Assign 3D array to a rank 3 tensor
- **D$IjkL = array_name[];** Assign 4D array to a rank 4 tensor

In using this syntax, the ‘array_name’ is the name of any properly dimension array in C. The dimension of the space and the rank of the tensor are used by the preprocessor to correctly assign the array elements to each component. For example, in the (x,y,z) Cartesian coordinate system, the assignment **B$Ij = matrix[];** would result in the following assignment ordering of the components with elements.

```
B$Xx = matrix[0][0];
B$Xy = matrix[0][1];
B$Xz = matrix[0][2];
                ...
B$Zy = matrix[2][1];
B$Zz = matrix[2][2];
```

The inverse is also allowed by simply turning the syntax of the assignment statement around, having a similar effect on the output.

- **array_name[] = A$i;** Assign vector to 1D array
- **array_name[] = B$Ij;** Assign rank 2 tensor to 2D array
- **array_name[] = C$ijK;** Assign rank 3 tensor to 3D array
- **array_name[] = D$IjkL;** Assign rank4 tensor to 4D array

It is also allowed to use a sub-range of a tensor in these assignment by placing a coordinate in the place of an index.

- **array_name[] = B$Iz;** Assign Xz, Yz, Zz components of the rank 2 tensor **B** to 1D array
- **C$xyK = array_name[];** Assign 1D array to xyX, xyY, xyZ components of the rank 3 tensor **C**
- **D$IxyL = array_name[];** Assign 2D array to XxyX, XxyY, XxyZ, ... , ZxyZ components of the rank 4 tensor **D**

Taken together, these allow for a very convenient interface to matrix manipulating functions that are available in numerical libraries available to scientific and engineering software developers.

Shorthand notation for rank1 assignments

There is a shorthand notation for making assignments to rank 1 tensors (vectors) which allows a single line assignment of N components, where N is the dimension of the space the tensors describe. In the Cartesian coordinate system that we have been using this would look something like this

- **Ro$i = {xo, yo, zo};** Assign xo to Ro$x, yo to Ro$y and zo to Ro$z
- **T$xi = {3.0, 2.0, sin(x)};** Assign 3.0 to T$xx, 2.0 to T$xy and sin(x) to T$xz
- **U$ii = {uxx, uyy, 1.0};** Assign uxx to U$xx, uyy to U$yy and 1.0 to U$zz

This syntax can be turned around just as with the array assignments. Of course the components within the { } must be scalar variables as below

- **{xo, yo, zo} = Ro$i;** Assign Ro$x to xo, Ro$y to yo and zo Ro$z to zo
- **{uxx, uyy, uzz} = U$ii;** Assign U$xx to uxx, U$yy to uyy and U$zz to uzz

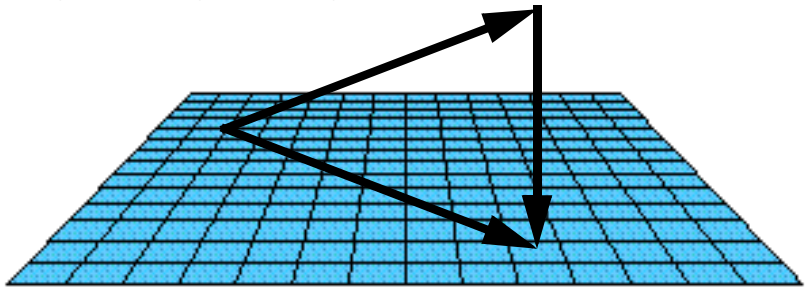
Each component within the { } must be separated by a comma and there must be the same number of components as the dimension that of the tensor's space. Extra spaces are also a good idea.

All of the examples shown have used three dimensional Cartesian space, but anything from 1 to 4 dimensions is allowed. Simply generalize the examples.

CHAPTER 4

Tensor Algebra

$$S^i_j = R^i_j + T^i_j \qquad T^j_k = A^j_i B^i_k$$



$$G_{jk} = R_{jk} - {}^1\!/_2\, g_{jk}\, R^i_i$$

Having the capability to assign values to tensor, the next requirement of a computer language for tensors would be use those values in combination, that is to say, to do tensor algebra. This entails tensor addition, subtraction, contractions and tensor products.

Arithmetic operations

The meaning of simple arithmetic operations such as addition, subtraction, multiplication and division as they apply to tensors are fully supported by GRPP. Tensor addition can be used to combine the values of components of two tensors individually so long as the tensors are of equal rank. This also applies to the operation of tensor subtraction. Here are some examples of the GRPP syntax used for these two operations.

- **A$Ij = B$Ij + C$Ij;** Add each component of tensor **B** to tensor **C** and assign the result to each component of tensor **A**

- **B$Ij = A$Ij - C$Ij;** Subtract each component of tensor **C** from tensor **A** and assign the result to each component of tensor **B**

Scalar multiplication with tensors is also accomplished in the same way.

- **A$Ij = 2.0 * B$Ij;** Multiple each component of tensor **B** by 2.0 and assign the result to each component of tensor **A**
- **B$Ij = - C$Ij / 3.0;** Divide the negative of each component of tensor **C** by 3.0 and assign the result to each component of tensor **B**

Arbitrarily complex expression involving addition, subtraction, scalar multiplication and scalar division can be built up from these operations on the right hand side of a tensor assignment. Parenthesis can be used to establish the order in which operations are applied.

- **A$Ij =5.7 * (B$Ij + C$Ij);** Sum each component of tensor **B** with tensor **C** and then multiply the sum by 5.7 assigning the result to each component of tensor **A**
- **B$Ij = A$Ij - C$Ij / 3.0;** Divide each component of tensor **C** by 3.0 and add this to each component of **A** assigning the result to each component of tensor **B**
- **A$jk = B$jk + C$jk - D$jk;** Add each component of tensor **B** with each component of tensor **C** and subtracting each component of tensor **D** before assigning the result to each component of tensor **A**
- **B$Ij = - C$Ij / 3.0;** Divide each component of tensor **C** by 3.0 and assign the result to each component of tensor **B**

The use of white spaces in-between tensors and operators increases readability and avoid unresolved parsing errors within the preprocessor. Their use is highly recommended in GRPP source code and must be use on each side of the * operator..

Tensor contractions

Summing over a contravariant and covariant index in a particular tensor results in a new tensor whose rank is reduced by 2 from the original tensor. This operation is called contraction. Here is a simple example of contraction of a rank 2 tensor to produce a scalar in 3 dimensional Cartesian coordinates.

$$R = R^{k}_{k} = R^{x}_{x} + R^{y}_{y} + R^{z}_{z}$$

Within the syntax of GRPP, the operation of contraction has the same basic format.

- **R = R$Ii ;** Contract the first contravariant index with the second covariant index in the tensor **R** and assign the resulting value to the scalar **R**
- **B$j = A$Iji;** Contract the first contravariant index with the third covariant index in the tensor **A** and assign the resulting value to the vector **B**
- **C$jk = B$ijkI;** Contract the first covariant index with the fourth contravariant index in the tensor **B** and assign the resulting value to the rank 2 tensor **C**
- **c = C$IjiJ / 3.0;** Contract the first contravariant index with the third covariant index and the second covariant index with the fourth contravariant index in the tensor **C** and assign the resulting value to the scalar **c**

Tensor products

Multiplying two tensors, one of rank n and the other of rank m results in a new tensor of rank (n+m). This is sometimes referred to as the outer product. GRPP syntax for tensor multiplications employs the * operator.

- **A$Ijk = B$Ij * C$k;** Multiple the rank 2 tensor **B** by the rank 1 tensor **C**, assigning the resulting rank 3 tensor to **A**
- **M$IjkL = F$I * E$jkL;** Multiple the rank 1 tensor **F** by the rank 3 tensor **E**, assigning the resulting rank 4 tensor to **M**
- **D$ijk = A$i * B$j * C$k;** Multiple the rank 1 tensor **A** by the rank 1 tensor **B** and the rank 1 tensor **C**, assigning the resulting rank 3 tensor to **A**

Any two tensors can be multiplied in this way as long as the resulting rank is less than four.

Tensor multiplication simultaneously combined with contraction is referred to as the inner product. This operation, like contraction reduces the rank of the resultant tensor by 2 from the combined rank of the multiplied tensors. The syntax for the inner product also uses the * operator. Again, remember not to use any tensors over rank 4 when applying this operation.

- **A$Ijk = B$IjL * C$kl;** Take the inner product of the rank 3 tensor **B** with the rank 2 tensor **C**, assigning the resulting rank 3 tensor to **A**
- **S$IjLm = U$IK * V$jkLm;** Take the inner product of the rank 2 tensor **U** with the rank 4 tensor **V**, assigning the resulting rank 4 tensor to **S**
- **dot = A$J * B$j;** Take the inner product of the rank 1 tensor **A** with the rank 1 tensor **B** and assigning the resulting scalar value to **dot**

Using the metric in an inner product raises or lowers indices. This is a means of exchanging contravariant and covariant indices in a tensor.

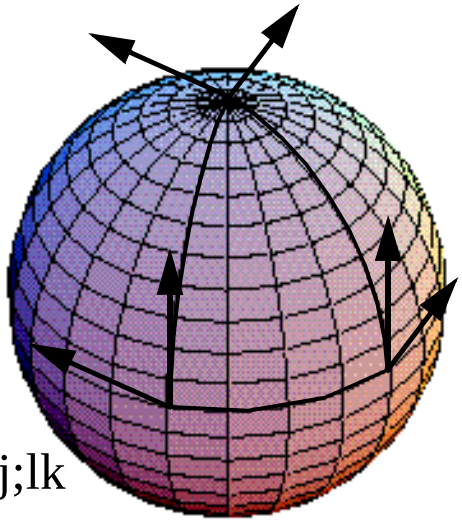
- **A$I = g$IJ * A$j;** Use the contravariant form of the metric **g** to raise the covariant index **j** of the rank 1 tensor **A**
- **S$ijk = g$jm * S$iMk;** Use the covariant form of the metric **g** to lower the contravariant index **M** of the rank 3 tensor **S**

Algebraic expression can be built up from these operators using parenthesis where needed to establish precedence. Expansion of these operation by the preprocessor results in very long expression. These expression can be difficult to follow so in general you should limit the number of operations performed in any GRPP syntax to make tracing and any possible debugging simpler. It may also be possible to create an algebraic expression that overflows internal buffers within GRPP. Here are some examples of more complex expressions combining many of these operations.

- **G$jk = R$jk - 0.5 * g$jk * R$Ii + lambda * g$jk;** Construct the rank 2 Einstein tensor **G**, cosmological scalar constant **lambda** included
- **D$Ijk = 0.5 * g$IL * (A$ljk + B$lkj - C$kj l);** Combine inner and scalar products with addition of tensor to produce the rank 3 tensor **D**
- **S$ijk = g$jm * S$iMk + A$i * B$jk;** Combine inner and outer products with addition of tensor to produce the rank 3 tensor **S**
- **a$I = - (C$Ijk * u$J * u$K);** Combine inner products with negation of resulting tensor to produce the vector **a**
- **S$ijk = (A$ijm + B$ijm) * (C$Mk + D$Mk);** Combine inner product with addition of tensors to produce the rank 3 tensor **S**
- **T$ij = (A$i* B$j) - (C$i * D$j);** Combine outer product with subtraction of tensors to produce the rank 2 tensor **T**
- **dT$ijx = (T_b$ij - T_a$ij) / dx;** Take finite difference of two rank 2 tensors scaled by infinitesimal **dx** and assign to the **ijx** components of the rank 3 tensor **dT**

CHAPTER 5

Tensor Calculus



$$R^{i}_{jkl}v_i = v_{j;kl} - v_{j;lk}$$

Physical laws in nature are representations of change. Differential equations are a familiar means of describing the behavior of changing physical quantities. This requires the use of calculus. Tensor calculus leads to differential geometry and require the use of derivatives of tensors. There are two fundamental types of derivatives used in tensor calculus, partial derivatives and covariant derivatives. Others can be built up from these. Partial and covariant derivatives of tensors can be used in algebraic tensor expressions just like any other tensors.

Partial derivatives

The partial derivative of the tensor $T_{ij,k}$ is a notational representation of the statement that the each component of the tensor T_{ij} is to have its partial derivative calculated with respect to the k^{th} coordinate. Here the comma is used to represent the taking of the partial derivative. GRPP syntax for tensors allows the use of the comma to mean partial derivative. In general, partial derivatives of tensors are not true tensors in the covariant sense.

- First order partial derivatives: A$,i and B$i,j and C$ij,k and D$ijk,l

- Second order partial derivatives: A$,Ij and B$i,jK and C$Ij,kL
- Third order partial derivatives: A$,ijk and B$i,jkl
- Fourth order partial derivatives: A$,ijkl

In each of these examples, the overall rank of the tensor was never greater than four. Indices were used on the previous examples. However, coordinates could have been freely mixed in.

- First order partial derivatives: A$,x and B$x,j and C$ij,y and D$xzz,y
- Second order partial derivatives: A$,Iy and B$i,yY and C$Ix,zZ
- Third order partial derivatives: A$,zyx and B$z,xyz
- Fourth order partial derivatives: A$,ijxy

Since the C and C++ languages do not all for the comma in variable names, the GRPP preprocessor must translate these into a string of characters that is both supported in these languages and unique in that the user would not be able to accidently declare the same string for another variable and confuse the two within the code. To do this the preprocessor prefixes a _d_ to first order partial derivatives, an _d2_ to second order partial derivatives, an _d3_ to third order partial derivatives and an _d4_ to fourth order partial derivatives. The comma character is then removed from the tensor name template to support syntax restriction within C and C++. As long as no variables are declared and used by the programmer with one of these prefixes, no conflict will be possible. To further avoid the possibility for this conflict, no tensor names in GRPP are allowed to begin with the underscore character.

Covariant derivatives

Covariant derivatives have a very similar notation. The semicolon is used in the place of the comma. The covariant derivative of the tensor $T_{ij;k}$ is a notational representation of the statement that the each component of the tensor T_{ij} is to have its covariant derivative calculated with respect to the k^{th} coordinate. The GRPP syntax allows the use of the semicolon to mean covariant derivative.

- First order covariant derivatives: A$;i and B$i;j and C$ij;k and D$ijk;l
- Second order covariant derivatives: A$;Ij and B$i;jK and C$Ij;kL
- Third order covariant derivatives: A$;ijk and B$i;jkl
- Fourth order covariant derivatives: A$;ijkl

Again, the overall rank of the tensor must never be greater than four. Coordinates can be use instead of indices just as with the partial derivatives.

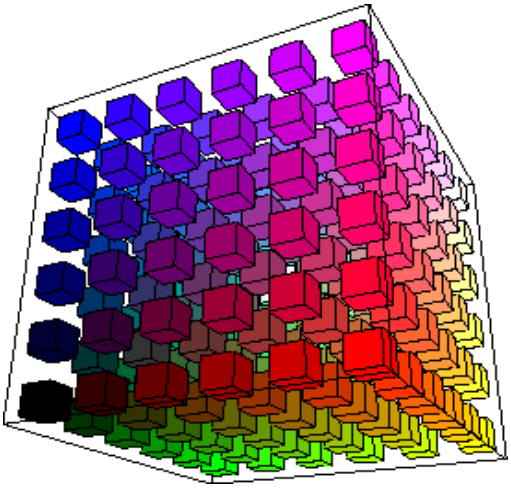
- First order covariant derivatives: A$;x and B$x;j and C$ij;y and D$xzz;y
- Second order covariant derivatives: A$;Iy and B$i;yY and C$Ix;zZ
- Third order covariant derivatives: A$;zyx and B$z;xyz
- Fourth order covariant derivatives: A$;ijxy

C and C++ also do not allow a semicolon in variable names, the GRPP preprocessor again translate these into a string of characters that are both supported in these languages and unique in that the user would not be able to accidently declare the same string for another variable and confuse the two within the code. To do this the preprocessor prefixes an _D_ to first order partial derivatives, an _D2_ to second order partial derivatives, an _D3_ to third order partial derivatives and an _D4_ to fourth order partial derivatives. Then the semicolon character is then removed from the tensor name template to support syntax restriction. As long as no variables are declared and used by the programmer with one of these prefixes, no conflict will be possible.

As was discussed in Chapter 2, when declaring tensors and partial derivatives and covariant derivatives of tensors, separate each by a comma followed by a space. This is to avoid a parsing conflict since the comma is being used to mean two things in these type casting declaration statements. It is being used as a separator and as a partial derivative character.

CHAPTER 6

Built-In Macros



Macros are building blocks of a computer code made up from the smaller units of the computer language. GRPP has several macros that perform complicated assignments common to tensor analysis.

Totally anti-symmetric tensor

GRPP has a built in statement to determine the Levi-Civita pseudo tensor in a Minkowskii or Cartesian coordinate system. The Levi-Civita pseudo tensor is defines such that

Cartesian Coordinates: $\varepsilon^{xyz} = 1$
Minkowskii Coordinates: $\varepsilon^{txyz} = 1$

Components derivable from an even permutation of these have a value of 1, odd permutations of these indices have a value of -1 and any components with repeated indices have a value of zero. The totally antisymmetric tensor of an arbitrary coordinate basis can be determined up to a sign from the Levi-Civita pseudo tensor. In order to define the Levi-Civita pseudo tensor in GRPP one uses the statement

```
$$
coordinates$(x,y,z);
indices$(i,j,k,l,m);
rank3 epsilon$IJK;
...
/* Set up totally antisymmetric tensor */
totally_antisymmetric$(epsilon);
...
$$
...
```

GRPP code BLOCK

Where epsilon is any legitimate tensor name prefix. One note of caution. The Levi-Civita pseudo tensor that this statement generates has a rank equal to the dimension defined by the coordinate$ statement. Since GRPP only supports up to rank 4 tensors, it is necessary to restrict use of this statement to manifolds of less than 4 dimensions.

Covariant differentiation

There is also a macro to have GRPP expand out the covariant derivative when working in a coordinate basis. The covariant derivative requires that the two geometric quantities, partial derivative and connection coefficients or Christoffel symbols be defined prior to its use. For example, if the covariant derivative $A^i{}_{;j}$ is to be calculated, the partial derivative $A^i{}_{,j}$ and the connection coefficients $\Gamma^i{}_{jk}$ are first needed. Assuming these are defined then the GRPP syntax for calculating $A^i{}_{;j}$ is the following:

```
$$
coordinates$(x,y,z);
indices$(i,j,k,l,m);
...
/* Set up covariant derivative tensor */
A$I;j = covariant_derivative$(A$I,j : Chr$Ijk);
...
$$
...
```

GRPP code BLOCK

The partial derivative and the connection coefficient are separated by a colon. This particular use of covariant_derivative$(:) ; is equivalent to the GRPP expression:

```
$$
coordinates$(x,y,z);
indices$(i,j,k,l,m);
...
/* Define the covariant derivative tensor */
A$I;j = A$I,j + Chr$Imk * A$M;
...
$$
...
```

GRPP code BLOCK

This example involved calculating the covariant derivative of a rank one tensor. As the rank of the tensor increases the expression for the covariant derivative becomes more complex. Here it was just as easy to write down the expression for the covariant derivative. In the Higher rank tensors it becomes more useful to have GRPP handle the bookkeeping through this macro.

Also, notice that the covariant derivative requires a dummy index in its definition' in this case, the index (m,M) was used. GRPP will use the last index from the indices$() statement that is not already in use to build the inner product. This macro can be applied to any tensor that preserves the current rank 4 limitations of the preprocessor. Because of the contraction, this means it can be used with rank 1 through 3.

Here is an example using covariant derivative of a rank 2 tensor. Notice the increasing complexity.

```
$$
coordinates$(x,y,z);
indices$(i,j,k,l,m);
...
/* Set up covariant derivative tensor */
A$Ij;k = covariant_derivative$(A$Ij,k : Chr$Ijk);
...
$$
...
```

GRPP code BLOCK

This is equivalent to the following line of GRPP code, but here the programmer is responsible for getting the correct covariant and contravariant indices summed over in the expression.

```
$$
coordinates$(x,y,z);
indices$(i,j,k,l,m);
...
/* Define the covariant derivative tensor */
A$Ij;k = A$Ij,k + Chr$Imk*A$Mj - Chr$Mjk*A$Im;
...
$$
...
```

GRPP code BLOCK

This illustrates the increasing complexity that results as the rank of the tensor being differentiated increases.

The observant reader may have noticed that the connection coefficient used in the covariant_derivative$(:) function was consistent. That is, a Chr$Ijk was used in the first and second example. The covariant derivative macro requires the use of a contravariant first index as with 'I' here. The macro will figure out all other indices to use based on the form of the first (partial derivative) first argument in the covariant derivative macro. The user must be responsible for having defined and assigned these connection coefficients and partial derivatives in earlier lines of GRPP code.

Lie Derivative

The Lie derivative, like the covariant derivative, plays an important role in differential geometry. It provides a measure of the change seen by an observer making and infinitesimal displacement in the direction of a vector v^i and carrying his coordinate system along with him. For a given vector field v^i the Lie derivative $\mathbf{\pounds}_{\boldsymbol{v}}$ of a tensor is expressed mathematically as

$$\mathbf{\pounds}_{\boldsymbol{v}} T_i = T_{i,m}\, v^m + T_m\, v^m{}_{,i}$$

$$\mathbf{\pounds}_{\boldsymbol{v}} T^i = T^i{}_{,m}\, v^m - T^m\, v^i{}_{,m}$$

$$\pounds_{\boldsymbol{v}} T_{ij} = T_{ij,m}\, v^{m} + T_{mj}\, v^{m}{}_{,i} + T_{im}\, v^{m}{}_{,j}$$

$$\pounds_{\boldsymbol{v}} T^{ij} = T^{ij}{}_{,m}\, v^{m} - T^{mj}\, v^{i}{}_{,m} - T^{im}\, v^{j}{}_{,m}$$

$$\pounds_{\boldsymbol{v}} T^{i}_{j} = T^{i}_{j,m}\, v^{m} - T^{m}_{j}\, v^{i}{}_{,m} + T^{i}_{m}\, v^{m}{}_{,j}$$

$$\pounds_{\boldsymbol{v}} T_{i}^{j} = T_{i}^{j}{}_{,m}\, v^{m} + T_{m}^{j}\, v^{m}{}_{,i} - T_{i}^{m}\, v^{j}{}_{,m}$$

and so forth as the rank of the tensor being differentiated increases. The syntax for the Lie derivative macro in GRPP is following.

```
$$
coordinates$(x,y,z);
indices$(i,j,k,l,m);
...
/* Set up Lie derivative of tensor */
L_T$jk = Lie_derivative$(v$I : T$jk);
...
$$
...
```

GRPP code BLOCK

The first argument of the Lie_derivative$(:) macro it the contravariant vector which specifies the direction to move. The second argument is the Tensor in which the Lie derivative is being applied. These two arguments must be separated by a colon. The indices of the second argument become the indices of the resulting tensor. The Lie derivative preserves the rank of the tensor, but because of the contraction, it is only possible to apply this macro to tensors up to rank 3 in GRPP code.

Symmetrization and anti-symmetrization

Any rank 2 tensor can be decomposed into the sum of a symmetric and antisymmetric tensor. Many of the physically important tensors belong to one or the other of the classes of symmetric or antisymmetric tensors. Because symmetries play such an important role in nature, the ability to construct either a symmetric tensor or an asymmetric tensor over a range of indices is important to tensor analysis. This construction is referred to as symmetrization or anti-symmetrization of a tensor. Symmetrization is accomplished by forming the sum of tensor components with all permutations of indices and dividing by the total number of permutations. Anti-

symmetrization is accomplished by adding tensor components with even permutations of indices and subtraction tensor components with odd permutations of indices and then dividing by the total number of indices. For a rank three tensor, the conventional notation for symmetrization would be

$$T_{(jkl)} = (T_{jkl} + T_{kjl} + T_{klj} + T_{lkj} + T_{ljk} + T_{jlk}) / 3!$$

and for anti-symmetrization

$$T_{[jkl]} = (T_{jkl} - T_{kjl} + T_{klj} - T_{lkj} + T_{ljk} - T_{jlk}) / 3!$$

These operations can be applied to a subset of indices as with

$$T_{(jk)l} = (T_{jkl} + T_{kjl}) / 2!$$

and with

$$T_{j[kl]} = (T_{jkl} - T_{jlk}) / 2!$$

The syntax of GRPP for symmetrization and anti-symmetrization of tensors differs slightly from the conventional notation. Symmetrization syntax is as follows.

- A$<ij> , B$<ijk> , B$<ij>k , B$i<jk> , C$<ijkl> , C$<ijk>l , C$i<jkl> , C$<ij>kl , C$i<jk>l and C$ij<kl>

Notice the use of < > instead of ( ). This differs from the conventional notation and is meant to avoid several parsing conflicts. Anti-symmetrization syntax is more in line with conventional notation.

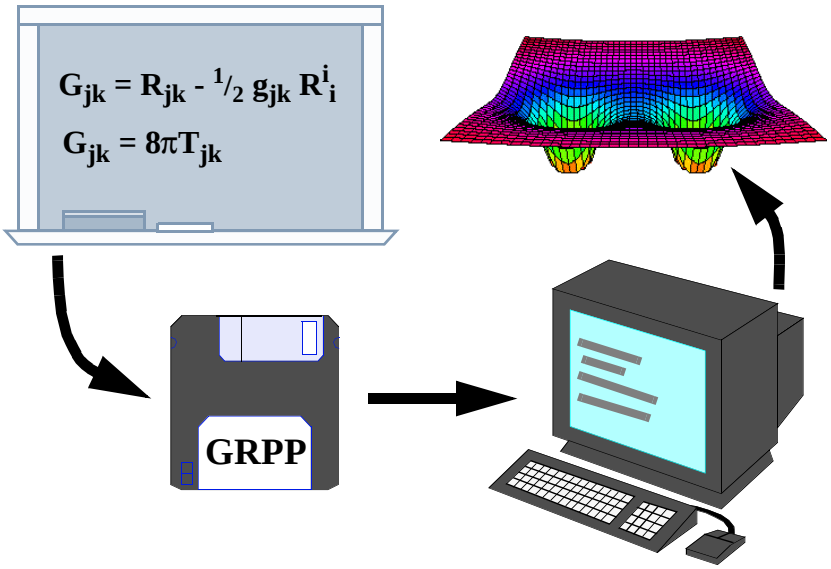
- A$[ij] , B$[ijk] , B$[ij]k , B$i[jk] , C$[ijkl] , C$[ijk]l , C$i[jkl] , C$[ij]kl , C$i[jk]l and C$ij[kl]

The GRPP implementation of symmetrization does not support the use of Bach Brackets. Indices that are being symmetrized or anti-symmetrized must be adjacent as in the examples above. It is a legal syntax to include partial and covariant derivatives of tensors in these operations.

- A$[i,j] , B$[ij;k] , B$[ij],k , B$i[;jk] , C$[ij,kl] , etc.

Covariant indices were used in all of these examples, but contravariant indices are also allowed. Symmetrized and anti-symmetrized tensors can be used like any other tensors in expressions.

CHAPTER 7 Usage

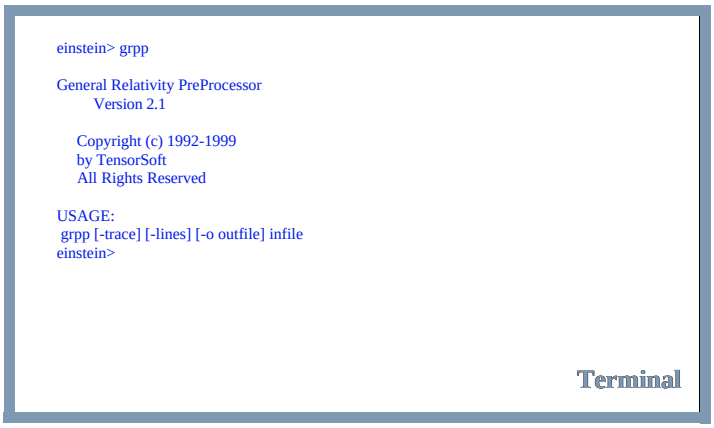


Having discussed the grammar of the GRPP tensor language, it is now time to demonstrate the user's interface to this preprocessor. GRPP is a preprocessor which translates the GRPP grammar into standard ANSI C code. It is run from the command line on the platforms the are supported. Complete development of an application program with GRPP requires that the computer platform have a C, Objective-C or C++ compiler installed. At present, GRPP is available only for UNIX systems, so one of these should be easily obtainable. You may wish to consider the gcc C compiler from GNU. The GRPP software package also comes with GRCC, a Bourne-shell script that will automatically invoke the C compiler and loader on your computer. This allows the developer to go directly from GRPP source code to an executable program with one command. The list of supported platform consist of the following at present.

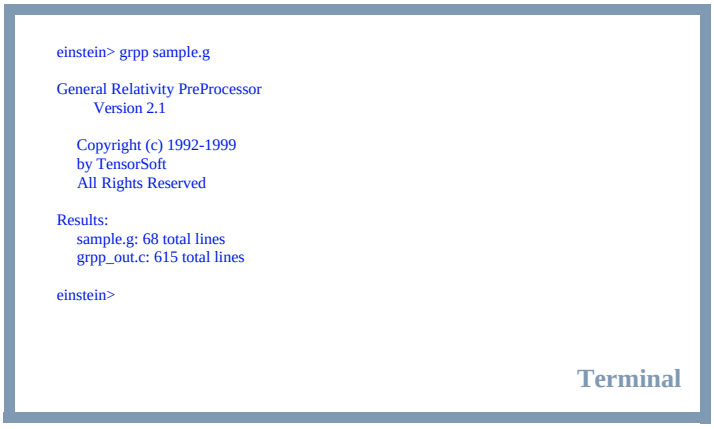
NeXT workstations running OpenStep 4.2
Sun Sparcstations running Solaris 5.6 & 5.7
Intel PCs running Linux 2.2
Alpha workstations running Linux 2.2
Windows9x using MS-DOS Prompt or CYGWIN B20.1

GRPP command line

The GRPP preprocessor is started up with the command, grpp in a terminal window. When no other inputs are specified, grpp will echo a usage message to the screen and then exit.



Command line options shown in [ ] are optional. The only required parameter for the preprocessor is the name of the input GRPP source code file. When the output file is not specified, grpp creates one with the name ‘grpp_out.c’, along with the associated header file ‘grpp_out.h’.



By specifying the name of the output file with the '-o outfile' command line option, the user can avoid overwriting existing files. This also forces the name of the header file generated by grpp to be '*outfile*_grpp.h' which is 'sample_grpp.h' in the following example. Appending the '_grpp.h' avoids stepping on any C header files the user may have written specifically for the output C code such as a 'sample.h' in this example.

```
einstein> grpp -o sample.c sample.g

General Relativity PreProcessor
        Version 2.1

    Copyright (c) 1992-1999
    by TensorSoft
    All Rights Reserved

Results:
    sample.g: 68 total lines
    sample.c: 615 total lines

einstein>
```

Terminal

The '-lines' option causes grpp to add information to the output C code about the line number in the of the GRPP source code that was used to produce the output lines of C code. This is particularly useful with ANSI C compilers which will reference the grpp code line number when an error in the C code syntax has occurred.

The '-trace' option causes grpp to printout tracing information as it preprocesses the GRPP source code. This is also useful in debugging. Consider the following short GRPP code example.

```
$$ /* set up coordinates and indices for GRPP */
coordinates$(x,y,z);
indices$(i,j,k,l);
$$

main()
{
 $$  /* Define tensors */
 rank2 A$Ij, B$ij, g$ij;
 B$ij = g$ij * A$Ij ;
 $$
}
```

GRPP code BLOCK

Using the '-trace' option produces the following output on at the terminal from grpp.

```
einstein> grpp -trace -o example.c example.g

General Relativity PreProcessor
        Version 2.1

    Copyright (c) 1992-1999
    by TensorSoft
    All Rights Reserved

Trace enabled
TRACE: A$Ij ->
      A_Cc$Ij
TRACE: B$ij ->
      B_cc$ij
TRACE: g$ij ->
      g_cc$ij
TRACE: B$ij ->
      B_cc$ij
TRACE: g$ij ->
      g_cc$ij
TRACE: g_cc$ij ->
      g_cc$ij
TRACE: A$Ij ->
      A_Cc$Ij
TRACE: A_Cc$Ij ->
      A_Cc$Ij
TRACE: g_cc$ij * A_Cc$Ij ->
      ( g_cc$xj * A_Cc$Xj + g_cc$yj * A_Cc$Yj + g_cc$zj * A_Cc$Zj )
TRACE: B_cc$ij = ( g_cc$xj * A_Cc$Xj + g_cc$yj * A_Cc$Yj + g_cc$zj * A_Cc$Zj )
Results:
    example.g: 32 total lines
    example.c: 53 total lines

einstein>
```

Terminal

Each step of the translation is echoed to the screen, preceded by ‘TRACE:’. The first 3 ‘TRACE:’ outputs are associated with the type casting ‘rank2’ statement in the source code. The next 7 ‘TRACE:’ outputs are all associated with translating the tensor equation. You should try this yourself using this very simple example. Also be sure to have a peek at the resulting C code produced by grpp which is now in the file ‘example.c’.

This can be a great help in matching up the last line preprocessed by GRPP when an error has occurred since the I/O buffers may not have been flushed to the output file when a statement can not be translated. It is also useful for watching the internal workings of grpp.

Errors which occur during the translation result in messages being sent to the screen, more specifically, to stderr. These error messages include the input file line number that was being translated and the output line number that would have been written. Errors also include a one or two line description of the nature of the error.

Using GRCC

The GRPP software also comes with a Bourne shell script which will make the pre-processing call to grpp and then pass the resulting file to the C compiler on your system. The usage of grcc is given in comments within the script. Some of these options may not apply to your system's compilers, assemblers and loaders.

```
#
#       GRCC script
#       Copyright (c) 1992-1999
#       by TensorSoft
#       All Rights Reserved
#
# cc-style script to compile and load GRPP, C, and assembly codes
#       usage:grcc [-O] [-o execfile] [-c] files [-L lib path] [-l library]
#       -O              optimize, passed to C compiler
#       -c              Do not call linker, leave relocatables in *.o
#       -o execfile     Override default executable name a.out
#       -S              leave assembler output on file.s
#       -l library      (passed to ld)
#       -L library      path(passed to ld)
#       -t              GRPP trace flag
#       files           GRPP source files ending in .g
#        "              C source files ending in .c
#        "              Assembly language files ending in .s
#       -D def          passed to C compiler (for .c files)
#       -I includepath  passed to C compiler (for .c files)
```

There are several lines within the grcc script which you may wish to tailor to you particular system configuration or personal needs. Among these are, which C compiler to use, where grcc should look for the grpp executable and what GRPP options to make defaults.

```
#set the C compiler to use
CC=${CC_grpp:-/bin/cc}
#Set the location of GRPP for your system below
#GRPP=${GRPP:-/usr/local/bin/grpp}
GRPP=${GRPP:-./grpp}
#Set any GRPP Flags that you wish to be defaults within GRCC
GRPPFLAGS=-lines
```

Reasonable values have been provided for all of these. For the location of grpp, the default is the local directory. However, you will probably wish to install grpp and grcc in a location that is in your shell's path. This will also make grpp and grcc available to other users on your system. It is recommended that your place grpp and grcc in the /usr/local/bin directory and add this directory to your shell's path. Then just remove the comment from the line '#GRPP=${GRPP:-/usr/local/bin/grpp}' and comment out the line 'GRPP=${GRPP:-./grpp}'. There is probably nothing else within this file that needs to be altered by the user.

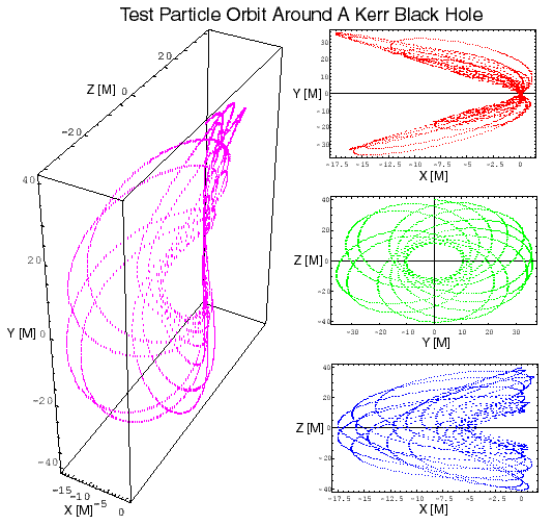
The following is an example of using grcc to translate and compile a sample GRPP program for making a single time step integration of the equations of motion of a test particle falling into a Schwarzschild black hole.

```
einstein> grcc -O -o sample sample.g

General Relativity PreProcessor
        Version 2.1

    Copyright (c) 1992-1999
    by TensorSoft
    All Rights Reserved

#Line enabled
Results:
    sample.g: 68 total lines
    sample.c: 683 total lines

einstein> ls
grcc*       sample*     sample.g    sample_grpp.h
grpp*       sample.c    sample.o
einstein> sample
time: 0.000000 radius: 10.000000 azimuth: 1.570785 polar: 0.000000
time: 0.100000 radius: 9.999920 azimuth: 1.570785 polar: 0.000000
einstein>
```

Terminal

This session used the defaults for grcc, which included the enabling of the #Line number C preprocessor directive. The files ‘sample.c’, ‘sample_grpp.h’, ‘sample.o’ and ‘sample’ were all created with this one command. The last command line runs the sample program which prints out the original position of the test particle and then the position one integration time step later.

CHAPTER 8 *Examples*



The above figures represent various projections of the orbits of a test particle about a rotating black hole. These projections are the results from integration of the fully relativistic four dimension equations of motion of the test particle about the Kerr black hole using GRPP. The source consisted of about 75 lines of GRPP code along with a couple of popular numerical library routines. Using the newer features in GRPP version 2.1 this code could have been reduced to about half that number of lines. The translated code contains over 750 lines of C code. That's a substantial savings in programming time and effort and the major motivation behind GRPP.

Using GRPP to tackle a problem like the one illustrated above is preceded by many questions. What types of problems are suited for GRPP? What features of the GRPP language will facilitate solving those problems suited for GRPP? How do I get started with GRPP? Questions like these usually imply a learning curve for the user. We learn best through a combination of examples and doing. This chapter contains several examples to help you on the way to better understanding and applying GRPP.

Imitation is said to be the *art of flattery* and necessity the mother of invention. Programming styles are very individual so don't feel you must stick to the formats demonstrated in the examples that follow. Experiment with the tools at hand and remember that GRPP is meant to make your job easier.

Crevasse deformation in an ice field

This first example is not from general relativity and is meant to demonstrate the usefulness of GRPP to other disciplines. It is also an example that is short enough to allow the translated code to be listed here. Problems that are associated with 3 and 4 dimensions tend to produce more lines of translated code as more components are defined.

This is an example from glaciology in which the stresses in an ice field are used to model crevasse deformation. It is a two dimensional problem using these simplifications. The stresses in the ice are given by the velocity gradients.

$$u^x{}_{,x} = \partial u^x/\partial x \equiv \textit{longitudinal stretching}$$
$$u^x{}_{,y} = \partial u^x/\partial y \equiv \textit{side shearing}$$
$$u^y{}_{,y} = \partial u^y/\partial y \equiv \textit{lateral extension}$$
$$u^y{}_{,x} = \partial u^y/\partial x \equiv \textit{flow-line turning}$$

Where $u^x = dx/dt$ and $u^y = dy/dt$. If reasonable values are assumed for these stresses, then the trajectories of points along a crevasse in the ice field are approximated by a first order Taylor series expansion about the origin of the velocity.

$$u^x(x,y) = u^x(0,0) + (u^x{}_{,x})\, x + (u^x{}_{,y})\, y$$
$$u^y(x,y) = u^y(0,0) + (u^y{}_{,x})\, x + (u^y{}_{,y})\, y$$

Or written as a single tensor equation,

$$u^i(x,y) = u^i(0,0) + (u^i{}_{,j})\, r^j$$

where $r^j = (x,y)$ is the position vector and can be solved by integration.

$$r^i = \int u^i \, dt$$

This is a set of coupled differential equation which can easily be integrated by the Euler Method given a set of initial conditions.

Here is the GRPP source code used to solve these differential equations using the Euler method to take steps forward in time.

```
/*
  File:example1.g
  Author:TensorSoft
  Date:December 1993
  Purpose:Two dimensional glacier ice flow

  Copyright (c) 1993-1999
```

```
  By TensorSoft
  All Rights Reserved
*/

$$ /* set up coordinate system and tensor indices for GRPP */
coordinates$(x,y);
indices$(i,j,k);
$$

#include <stdio.h>
#include <math.h>

#define SECONDS_PER_MONTH 2628000
#define SECONDS_PER_YEAR  31536000
#define TIME_STEP         100

static char *mon[12] = {"Jan","Feb","Mar","Apr",
                        "May","Jun","Jul","Aug",
                        "Sep","Oct","Nov","Dec"};

main()
{
 int month,seconds;
 double t,x,y,ux,uy,dt = ((double)TIME_STEP / (double)SECONDS_PER_YEAR),
                  x_o = 0.0,
                  y_o = 0.0,
                  ux_o = 0.0,
                  uy_o = 0.05,
                  longitudinal_stretching = 0.0001,
                  side_shearing = 0.1,
                  lateral_extension = 0.0001,
                  flowline_turning = 0.1;

$$ /* Define tensors */
rank1 r$I, u_o$i, u$i, u$I;
rank2 eta$IJ, u$i,j;
$$

$$ /* initialize tensors */
 r$I = {x_o, y_o};
 u_o$i = {ux_o, uy_o};
 u$x,x = longitudinal_stretching;
 u$x,y = side_shearing;
 u$y,x = flowline_turning;
 u$y,y = lateral_extension;
 eta$XY =+= 0.0;
 eta$XX = 1.0;
 eta$YY = 1.0;
$$
 x = x_o; y = y_o; ux = ux_o; uy = uy_o;
 printf("Initial:    position[km](%f,%f) velocity[km/yr](%f,%f) \n",x,y,ux,uy);
 for (month=0; month<12; month++) {
   for (seconds=0; seconds<SECONDS_PER_MONTH; seconds+=TIME_STEP) {
     $$ /* integrate */
     u$i = u_o$i + u$i,j * r$J;
     r$I += eta$IJ * u$j * dt;
     $$
     }
   $$ /* place monthly results in ANSI C variables for printing */
   {x,y} = r$I;
   {ux,uy} = u$i;
   $$
 printf("End of %s: position[km](%f,%f) velocity[km/yr](%f,%f) \n",mon[month],x,y,ux,uy);
 }
}
```

This can now be preprocessed using grpp or preprocessed and compiled using grcc. Here is an example of a session using grcc on this source code.

```
einstein> grcc -O -o example1 example1.g

General Relativity PreProcessor
        Version 2.1

    Copyright (c) 1992-1999
    by TensorSoft
    All Rights Reserved

#Line enabled
Results:
    example1.g: 73 total lines
    example1.c: 186 total lines

einstein> ls
example1*       example1.g      example1_grpp.h  grpp*
example1.c      example1.o      grcc*
einstein> example1
Initial:    position[km](0.000000,0.000000) velocity[km/yr](0.000000,0.050000)
End of Jan: position[km](0.000017,0.004167) velocity[km/yr](0.000417,0.050002)
End of Feb: position[km](0.000069,0.008334) velocity[km/yr](0.000833,0.050008)
End of Mar: position[km](0.000156,0.012501) velocity[km/yr](0.001250,0.050017)
End of Apr: position[km](0.000278,0.016670) velocity[km/yr](0.001667,0.050029)
End of May: position[km](0.000434,0.020840) velocity[km/yr](0.002084,0.050045)
End of Jun: position[km](0.000625,0.025011) velocity[km/yr](0.002501,0.050065)
End of Jul: position[km](0.000851,0.029184) velocity[km/yr](0.002918,0.050088)
End of Aug: position[km](0.001112,0.033359) velocity[km/yr](0.003336,0.050114)
End of Sep: position[km](0.001407,0.037537) velocity[km/yr](0.003754,0.050144)
End of Oct: position[km](0.001737,0.041717) velocity[km/yr](0.004172,0.050178)
End of Nov: position[km](0.002102,0.045900) velocity[km/yr](0.004590,0.050215)
End of Dec: position[km](0.002502,0.050086) velocity[km/yr](0.005009,0.050255)
einstein>
```

Terminal

There are four files produced by grcc; example1, example1.o and example1.c. The executable, example1 was run and the results are given above. The translated files, example1_grpp.h and example1.c are the output from grpp. Here is the listing for example1.c after being reprocessed with ‘grpp -o example1.c example1.g’ to remove the #line preprocessor directives for greater readability.

```
/*
  File:example1.g
  Author:TensorSoft
  Date:December 1993
  Purpose:Two dimensional glacier ice flow

  Copyright (c) 1993-1999
  By TensorSoft
  All Rights Reserved
*/
```

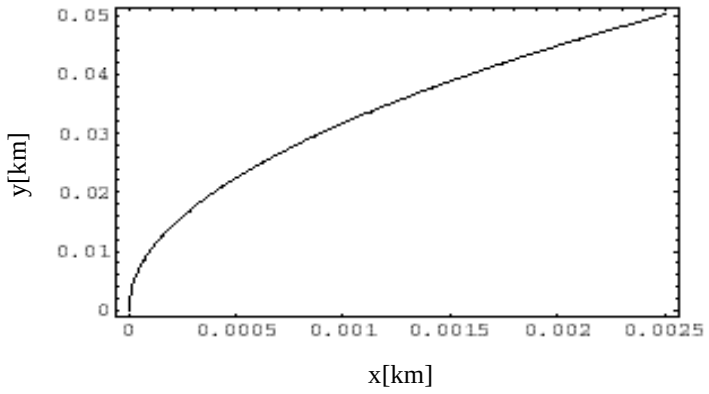
```
/* Open  GRPP Block */
/* set up coordinate system and tensor indices for GRPP */
/* COORDINATES: XxYy */
/* DIMENSION: 2 */

#include "example1_grpp.h"

/* INDICES: IiJjKk */
/* TOTAL: 3 */

/* Close GRPP Block */

#include <stdio.h>
#include <math.h>

#define SECONDS_PER_MONTH 2628000
#define SECONDS_PER_YEAR  31536000
#define TIME_STEP         100

static char *mon[12] = {"Jan","Feb","Mar","Apr",
                        "May","Jun","Jul","Aug",
                        "Sep","Oct","Nov","Dec"};

main()
{
 int month,seconds;
 double t,x,y,ux,uy,dt = ((double)TIME_STEP / (double)SECONDS_PER_YEAR),
                  x_o = 0.0,
                  y_o = 0.0,
                  ux_o = 0.0,
                  uy_o = 0.05,
                  longitudinal_stretching = 0.0001,
                  side_shearing = 0.1,
                  lateral_extension = 0.0001,
                  flowline_turning = 0.1;


/* Open  GRPP Block */
/* Define tensors */
 TENSOR_C r_C ;
 TENSOR_c u_o_c ;
 TENSOR_c u_c ;
 TENSOR_C u_C ;

 TENSOR_CC eta_CC ;
 TENSOR_cc _d_u_cc ;

/* Close GRPP Block */


/* Open  GRPP Block */
/* initialize tensors */
 r_C.X = x_o;
 r_C.Y =  y_o;

 u_o_c.x = ux_o;
 u_o_c.y =  uy_o;

 _d_u_cc.xx = longitudinal_stretching;

 _d_u_cc.xy = side_shearing;

 _d_u_cc.yx = flowline_turning;

 _d_u_cc.yy = lateral_extension;

 eta_CC.YX = ( eta_CC.XY = 0.0 );
```

```
 eta_CC.XX = 1.0;

 eta_CC.YY = 1.0;

/* Close GRPP Block */
 x = x_o; y = y_o; ux = ux_o; uy = uy_o;
 printf("Initial:    position[km](%f,%f) velocity[km/yr](%f,%f) \n",x,y,ux,uy);
 for (month=0; month<12; month++) {
    for (seconds=0; seconds<SECONDS_PER_MONTH; seconds+=TIME_STEP) {

/* Open  GRPP Block */
/* integrate */
 u_c.x = u_o_c.x + ( _d_u_cc.xx * r_C.X + _d_u_cc.xy * r_C.Y );
 u_c.y = u_o_c.y + ( _d_u_cc.yx * r_C.X + _d_u_cc.yy * r_C.Y );

 r_C.X += ( eta_CC.XX * ( u_c.x * dt ) + eta_CC.XY * ( u_c.y * dt ) );
 r_C.Y += ( eta_CC.YX * ( u_c.x * dt ) + eta_CC.YY * ( u_c.y * dt ) );

/* Close GRPP Block */
       }

/* Open  GRPP Block */
/* place monthly results in ANSI C variables for printing */
 x = r_C.X;
 y = r_C.Y;

 ux = u_c.x;
 uy = u_c.y;

/* Close GRPP Block */
 printf("End of %s: position[km](%f,%f) velocity[km/yr](%f,%f) \n",mon[month],x,y,ux,uy);
 }
}
```

Compare this example1.c file with the example1.g source code to see the action of the preprocessor. You may wish to run ‘grcc’ or ‘grpp -lines’ to match up the input and output.

The results can be plotted to give a pattern of the deformation of the crevasse over the course to the year for which it was integrated in this example.

Crevasse Deformation after one year



Free fall into a Schwarzschild black hole

This example is from general relativity. A test particle is released from rest at a distance R in Boyer-Lindquist coordinates from the center of a Schwarzschild black hole. To an observer at infinity, the test particle takes an infinite amount of time to cross over the event horizon. However, in the rest frame of the test particle, the time to cross the event horizon is finite. The proper time and coordinate times can be determined by integrating the geodesic equation for a free falling test particle starting from rest at a distance R from the origin.

The metric for the Schwarzschild black hole is given by

$$ds^2 = -(1/(1-2M/r))dt^2 + (1+2M/r)dr^2 + r^2d\theta^2 + r^2\sin^2\theta d\phi^2$$

where M is the mass of the black hole. Geometrized units (G=c=1) are used throughout examples from general relativity. The geodesics determine the equations of motion for the test particle and are given from differential geometry by

$$d^2x^i/d\tau^2 = -\Gamma^i_{jk}\,(dx^j/d\tau)\,(dx^k/d\tau)$$

The Γ^i_{jk} are the Christoffel symbols and are defined by

$$\Gamma^i_{jk} = {}^1/_2\, g^{il}(\, g_{lj,k} + g_{lk,j} - g_{kj,l}\,)$$

where the g^{il} are the contravariant components of the metric and the $g_{ij,k}$are the partial derivatives of the covariant components of the metric.

A simple Euler method for integrating the geodesic equations will be used. This introduces rather large numerical error. But employing a more robust differential equation solver like the Runge-Kutta would over shadow the simplicity of the GRPP source code at this time. Here is the GRPP source code for this example. It will integrate forward along the free falling test particle's geodesic until the event horizon is crossed.

```
/*
  File:example2.g
  Author:TensorSoft
  Date:December 1993
  Purpose:Test particle free fall into Schwarzchild black hole

  Copyright (c) 1993-1999
  By TensorSoft
  All Rights Reserved
*/

$$ /* set up coordinate system and tensor indices for GRPP */
coordinates$(t,r,a,p);
indices$(i,j,k,l,m,n);
$$
```

```
#include <stdio.h>
#include <math.h>

main()
{
 int i,j,k,line,nsteps;
 double t,r,a,p,ut,ur,ua,up;
 double sin_a,cos_a,R,M,tau,dtau;
 double g_array[4][4];

 $$  /* Define tensors */
 rank1 r$I, u$I, accel$I, diag$i, diag$I;
 rank2 g$ij, g$IJ;
 rank3 g$ij,k, Chris$Ijk;
 $$

 M = 1;
 tau = t = 0;ut = 1;
 r = 8.0;ur = 0;
 a = 3.14157/2.0;ua = 0;
 p = 0;up = 0;
 sin_a = sin(a);
 cos_a = cos(a);
 dtau = 1.0e-6;
 printf("Tp: %f Tc: %f radius: %f theta: %f phi: %f\n",tau,t,r,a,p);
$$
 g$ij = 0;
 g$IJ = 0;
 g$ij,k = 0;
$$
 for (line=0; line<26; line++) {
 for (nsteps=0;nsteps<1000000;nsteps++) {
 $$
 r$I = {t,r,a,p};
 u$I = {ut,ur,ua,up};
 diag$i = { -(1.0-2.0*M/r), 1.0/(1.0-2.0*M/r), r*r, r*r*sin_a*sin_a };
 g$ii = diag$i;
 diag$I = {-1.0/(1.0-2.0*M/r),(1.0-2.0*M/r),1.0/(r*r),1.0/(r*r*sin_a*sin_a)};
 g$II = diag$I;
 g$tt,r = - 2.0 * M / ( r * r );
 g$rr,r = 2.0 * M / (( r - 2.0 * M ) * ( r - 2.0 * M ));
 g$aa,r = 2.0 * r;
 g$pp,r = 2.0 * r * sin_a * sin_a;
 g$pp,a = 2.0 * r * r * sin_a * cos_a;
 Chris$Ijk = 0.5 * g$IL * ( g$lj,k + g$lk,j - g$kj,l );
 accel$I = - ( Chris$Ijk * u$J * u$K );
 u$I += accel$I * dtau;
 r$I += u$I * dtau;
 {t,r,a,p} = r$I;
 {ut,ur,ua,up} = u$I;
$$
 tau += dtau;
 if (r<=(2*M))
   {
    printf("Passed through event horizon at proper time: %f \n",tau);
    printf("                    and at coordinate time: %f \n",t);
    exit(0);
   }
 }
 printf("Tp: %f Tc: %f radius: %f theta: %f phi: %f\n",tau,t,r,a,p); }
}
```

Preprocessing this source code with grpp produces this output at the terminal. Which can then be compiled with the C compiler.

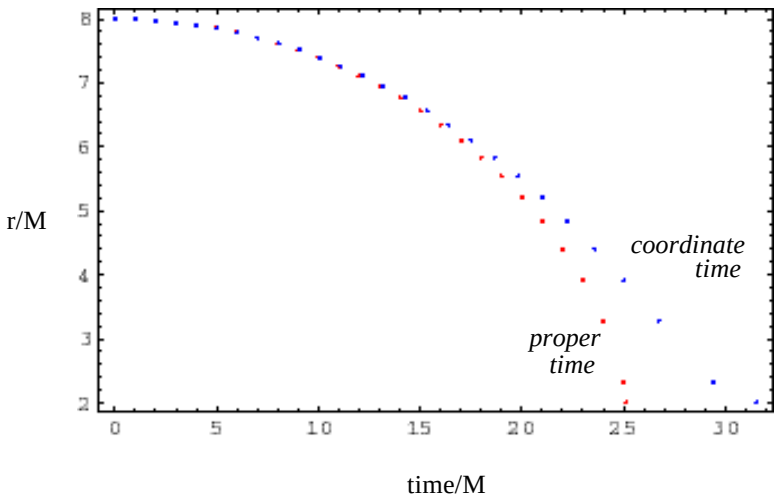
```
einstein> grpp -o example2.c example2.g

General Relativity PreProcessor
        Version 2.1

    Copyright (c) 1992-1999
    by TensorSoft
    All Rights Reserved

Results:
    example2.g: 78 total lines
    example2.c: 601 total lines

einstein> cc -O -o example2 example2.c
einstein> ls
example2*       example2.g      example2_grpp.h  grpp*
example2.c      example2.o      grcc*
einstein> example2
Tp: 0.000000 Tc:  0.000000 radius: 8.000000 theta: 1.570785 phi: 0.000000
Tp: 1.000000 Tc:  1.000081 radius: 7.994139 theta: 1.570785 phi: 0.000000
Tp: 2.000000 Tc:  2.000653 radius: 7.976532 theta: 1.570785 phi: 0.000000
Tp: 3.000000 Tc:  3.002213 radius: 7.947110 theta: 1.570785 phi: 0.000000
Tp: 4.000000 Tc:  4.005274 radius: 7.905756 theta: 1.570785 phi: 0.000000
Tp: 5.000000 Tc:  5.010376 radius: 7.852302 theta: 1.570785 phi: 0.000000
Tp: 6.000000 Tc:  6.018091 radius: 7.786525 theta: 1.570785 phi: 0.000000
Tp: 7.000000 Tc:  7.029037 radius: 7.708143 theta: 1.570785 phi: 0.000000
Tp: 8.000000 Tc:  8.043894 radius: 7.616807 theta: 1.570785 phi: 0.000000
Tp: 9.000000 Tc:  9.063415 radius: 7.512091 theta: 1.570785 phi: 0.000000
Tp: 10.00000 Tc: 10.088455 radius: 7.393481 theta: 1.570785 phi: 0.000000
Tp: 11.00000 Tc: 11.119993 radius: 7.260359 theta: 1.570785 phi: 0.000000
Tp: 12.00000 Tc: 12.159171 radius: 7.111978 theta: 1.570785 phi: 0.000000
Tp: 13.00000 Tc: 13.207343 radius: 6.947436 theta: 1.570785 phi: 0.000000
Tp: 14.00000 Tc: 14.266146 radius: 6.765639 theta: 1.570785 phi: 0.000000
Tp: 15.00000 Tc: 15.337598 radius: 6.565239 theta: 1.570785 phi: 0.000000
Tp: 16.00000 Tc: 16.424241 radius: 6.344561 theta: 1.570785 phi: 0.000000
Tp: 17.00000 Tc: 17.529369 radius: 6.101480 theta: 1.570785 phi: 0.000000
Tp: 18.00000 Tc: 18.657385 radius: 5.833247 theta: 1.570785 phi: 0.000000
Tp: 19.00000 Tc: 19.814407 radius: 5.536187 theta: 1.570785 phi: 0.000000
Tp: 20.00000 Tc: 21.009368 radius: 5.205202 theta: 1.570785 phi: 0.000000
Tp: 21.00000 Tc: 22.256216 radius: 4.832813 theta: 1.570785 phi: 0.000000
Tp: 22.00000 Tc: 23.578881 radius: 4.407156 theta: 1.570785 phi: 0.000000
Tp: 23.00000 Tc: 25.024967 radius: 3.906920 theta: 1.570785 phi: 0.000000
Tp: 24.00000 Tc: 26.719028 radius: 3.283954 theta: 1.570785 phi: 0.000000
Tp: 25.00000 Tc: 29.455787 radius: 2.318731 theta: 1.570785 phi: 0.000000
Passed through event horizon at proper time: 25.132744
                and at coordinate time: 31.545474
einstein>
```

Terminal

The user is referred to the GRPP distribution disk for a look at the 601 lines of output in this example. Also, check the example2_grpp.h header file while you are looking at example2 code.

The results of integrating with the Euler method show a marked deviation from the analytic results as the test particle approaches the event horizon. This is to be expected from such a crude integration technique, but the following plot of these results does indicate the trend in the proper time and coordinate time are the expected ones.

Free fall into a Schwarzschild black hole



A charged particle in an electromagnetic field

The covariant description of electromagnetism employees a rank 2 antisymmetric tensor F_{ij} in 4 dimensional space-time. If the electromagnetic field strengths are sufficiently small and no gravitational fields are present, then the metric η^{il} is that of flat space-time. In such a situation as this, the equations of motion of a point charge are given by the tensor equation

$$m d^2x^i/d\tau^2 = q\, F^i_{\ j}\, (dx^j/d\tau)$$

where

$$F^i_{\ j} = \eta^{il}\, F_{lj}$$

$$\text{Diagonal}\{\eta^{ij}\} = \{-1,1,1,1\}$$

In Minkowskii coordinates the tensor F_{ij} has components

$$F_{tt} = F_{xx} = F_{yy} = F_{zz} = 0$$
$$-F_{tx} = F_{xt} = E_x$$
$$-F_{ty} = F_{yt} = E_y$$
$$-F_{tz} = F_{zt} = E_z$$

$$F_{xy} = -F_{yx} = B_z$$
$$F_{zx} = -F_{xz} = B_y$$
$$F_{yz} = -F_{zy} = B_x$$

In this example two features of the GRPP language are being demonstrated; the antisymmetric syntax, and the passing of tensors between functions. In this particular example the acceleration function is passed two rank 1 tensors for the velocity and the acceleration. The velocity is passed as a tensor structure and the acceleration is passed as a pointer to a tensor structure since its values are to be modified in this function. Here is the source code.

```
/*
  File:example3.g
  Author:TensorSoft
  Date:December 1993
  Purpose:Charged particle in Electromagnetic field

  Copyright (c) 1993-1999
  By TensorSoft
  All Rights Reserved
*/

$$ /* set up coordinate system and tensor indices for GRPP */
coordinates$(t,x,y,z);
indices$(i,j,k,l,m,n);
$$

#include <stdio.h>
#include <math.h>

#define PI     3.14159265359
#define MASS   1.0
#define CHARGE 1.0
#define FREQ   100

main()
{
 int nprint,isteps;
 double t,x,y,z,ut,ux,uy,uz,tau,dtau;
 void acceleration();
 $$ rank1 r$I, u$I, a$I; $$

 dtau = (1.0 / FREQ) / 100000;
 t = x = y = z = 0;
 ux = uy = uz = 0; ut = 1;
 $$
 r$I = {t,x,y,z};
 u$I = {ut,ux,uy,uz};
 $$
 printf(" %f %f %f %f %f \n",tau, t, x, y, z);
 for (nprint=0; nprint<100; nprint++)
   {
    for (isteps=0; isteps<2000; isteps++)
      {
       acceleration( t, u_C, &a_C );
       $$
       u$I += a$I * dtau;
       r$I += u$I * dtau;
       t = r$T;
       $$
       tau += dtau;
      }
    $$
    {ut,ux,uy,uz} = u$I;
```

```
    {t,x,y,z} = r$I;
    $$
    printf(" %f %f %f %f %f \n",tau, t, x, y, z);
   }
}

void acceleration( t, u_C, a_C )
double t;
$$ rank1 u$I, *a$I; $$
{
 double q,m,omega,phi,
        E_o,E_x,E_y,E_z,
        B_o,B_x,B_y,B_z;
$$
 rank2 Faraday$Ij;
$$

 omega = 2 * PI * FREQ;
 phi = PI / 2;
 q = CHARGE; m = MASS;
 E_o = 10000;
 B_o = 10;
 E_z = 0;
 B_z = 0;
$$ /* Initialize the tensors */
 Faraday$Ii = 0;
 Faraday$Zt =+=  E_z;
 Faraday$Yx =-= -B_z;
$$
 E_x =  E_o * cos( omega * t );
 B_y =  B_o * cos( omega * t );
 E_y =  E_o * cos( 2 * omega * t + phi);
 B_x =  B_o * cos( 2 * omega * t + phi);
$$
 Faraday$Xt =+=  E_x;
 Faraday$Yt =+=  E_y;
 Faraday$Zx =-=  B_y;
 Faraday$Zy =-= -B_x;
 *a$I = (q/m) * Faraday$Ij * u$J;
$$
}
```

Using grcc to translate and compile the source code results in the following terminal session.

```
einstein> grcc -O -o example3 example3.g

General Relativity PreProcessor
        Version 2.1

    Copyright (c) 1992-1999
    by TensorSoft
    All Rights Reserved

#Line enabled
Results:
    example3.g: 95 total lines
    example3.c: 167 total lines

einstein>
```

Terminal

There is very little tensor algebra in this example, resulting in a relatively small translated C code output. Here is the translated code found in the file example3.c.

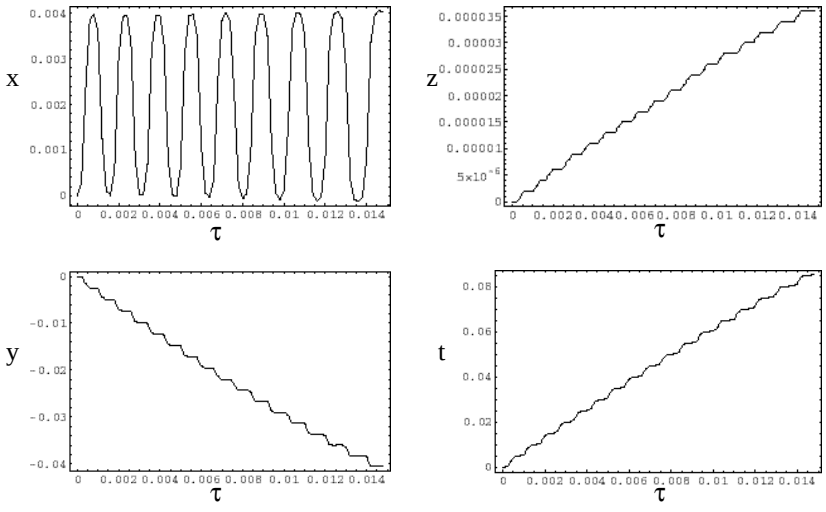
```
/*
  File:example3.g
  Author:TensorSoft
  Date:December 1993
  Purpose:Charged particle in Electromagnetic field

  Copyright (c) 1993-1999
  By TensorSoft
  All Rights Reserved
*/


/* Open  GRPP Block */
/* set up coordinate system and tensor indices for GRPP */
/* COORDINATES: TtXxYyZz */
/* DIMENSION: 4 */

#include "example3_grpp.h"

/* INDICES: IiJjKkLlMmNn */
/* TOTAL: 6 */

/* Close GRPP Block */

#include <stdio.h>
#include <math.h>

#define PI      3.14159265359
#define MASS    1.0
#define CHARGE  1.0
#define FREQ    100

main()
{
 int nprint,isteps;
 double t,x,y,z,ut,ux,uy,uz,tau,dtau;
 void acceleration();

/* Open  GRPP Block */
 TENSOR_C r_C ;
 TENSOR_C u_C ;
 TENSOR_C a_C ;
/* Close GRPP Block */

 dtau = (1.0 / FREQ) / 100000;
 t = x = y = z = 0;
 ux = uy = uz = 0; ut = 1;

/* Open  GRPP Block */

 r_C.T = t;
 r_C.X = x;
 r_C.Y = y;
 r_C.Z = z;

 u_C.T = ut;
 u_C.X = ux;
 u_C.Y = uy;
 u_C.Z = uz;

/* Close GRPP Block */
 printf(" %f %f %f %f %f \n",tau, t, x, y, z);
 for (nprint=0; nprint<100; nprint++)
    {
    for (isteps=0; isteps<2000; isteps++)
```

```
        {
        acceleration( t, u_C, &a_C );

/* Open  GRPP Block */

 u_C.T += ( a_C.T * dtau );
 u_C.X += ( a_C.X * dtau );
 u_C.Y += ( a_C.Y * dtau );
 u_C.Z += ( a_C.Z * dtau );

 r_C.T += ( u_C.T * dtau );
 r_C.X += ( u_C.X * dtau );
 r_C.Y += ( u_C.Y * dtau );
 r_C.Z += ( u_C.Z * dtau );

 t = r_C.T;

/* Close GRPP Block */
         tau += dtau;
        }

/* Open  GRPP Block */

 ut = u_C.T;
 ux = u_C.X;
 uy = u_C.Y;
 uz = u_C.Z;

 t = r_C.T;
 x = r_C.X;
 y = r_C.Y;
 z = r_C.Z;

/* Close GRPP Block */
     printf(" %f %f %f %f %f \n",tau, t, x, y, z);
    }
}

void acceleration( t, u_C, a_C )
double t;

/* Open  GRPP Block */
 TENSOR_C u_C ;
 TENSOR_C *a_C ;
/* Close GRPP Block */
{
 double q,m,omega,phi,
        E_o,E_x,E_y,E_z,
        B_o,B_x,B_y,B_z;

/* Open  GRPP Block */

 TENSOR_Cc Faraday_Cc ;

/* Close GRPP Block */

 omega = 2 * PI * FREQ;
 phi = PI / 2;
 q = CHARGE; m = MASS;
 E_o = 10000;
 B_o = 10;
 E_z = 0;
 B_z = 0;

/* Open  GRPP Block */
/* Initialize the tensors */
 Faraday_Cc.Tt = 0;
 Faraday_Cc.Xx = 0;
 Faraday_Cc.Yy = 0;
 Faraday_Cc.Zz = 0;
```

```
 Faraday_Cc.Tz = ( Faraday_Cc.Zt = E_z );

 Faraday_Cc.Xy = -( Faraday_Cc.Yx = -B_z );

/* Close GRPP Block */
 E_x =  E_o * cos( omega * t );
 B_y =  B_o * cos( omega * t );
 E_y =  E_o * cos( 2 * omega * t + phi);
 B_x =  B_o * cos( 2 * omega * t + phi);

/* Open  GRPP Block */

 Faraday_Cc.Tx = ( Faraday_Cc.Xt = E_x );

 Faraday_Cc.Ty = ( Faraday_Cc.Yt = E_y );

 Faraday_Cc.Xz = -( Faraday_Cc.Zx = B_y );

 Faraday_Cc.Yz = -( Faraday_Cc.Zy = -B_x );

 (*a_C).T = ( ( q / m ) * ( Faraday_Cc.Tt * u_C.T + Faraday_Cc.Tx * u_C.X
          + Faraday_Cc.Ty * u_C.Y + Faraday_Cc.Tz * u_C.Z ) );
 (*a_C).X = ( ( q / m ) * ( Faraday_Cc.Xt * u_C.T + Faraday_Cc.Xx * u_C.X
          + Faraday_Cc.Xy * u_C.Y + Faraday_Cc.Xz * u_C.Z ) );
 (*a_C).Y = ( ( q / m ) * ( Faraday_Cc.Yt * u_C.T + Faraday_Cc.Yx * u_C.X
          + Faraday_Cc.Yy * u_C.Y + Faraday_Cc.Yz * u_C.Z ) );
 (*a_C).Z = ( ( q / m ) * ( Faraday_Cc.Zt * u_C.T + Faraday_Cc.Zx * u_C.X
          + Faraday_Cc.Zy * u_C.Y + Faraday_Cc.Zz * u_C.Z ) );

/* Close GRPP Block */
}
```

Again, this example is using the Euler method to integrate the differential equations which will give less accurate results than one typically desires. But it serves the demonstration well. Here are plots of the coordinates of the charged particle as a function of proper time in this example.

Position of charged particle as a function of proper time



Curvature in equatorial plane of a Kerr black hole

This example is also from general relativity. The scalar curvature of space-time in the equatorial plane of a Kerr (rotating) black hole is numerically calculated from the Kerr metric. Since the Kerr metric is a vacuum solution to Einstein's field equations, the curvature should be zero. Any non-zero values found will be the result of numerical errors. Thus it is a useful test of the kind of accuracy that is possible with numerical relativity.

The Kerr metric is given by

$$ds^2 = -(1-2Mr/\Sigma)dt^2 - (4aMr\sin^2\theta)dtd\phi + (\Sigma/\Delta)dr^2 + (\Sigma^2)d\theta^2 + ((r^2+a^2 + 2Mra^2\sin^2\theta/\Sigma)\sin^2\theta)d\phi^2$$

where

$$a = J/M$$
$$\Delta = r^2 - 2Mr + a^2$$
$$\Sigma = r^2 + a^2\cos^2\theta$$

and M is the mass of the black hole, J is the angular momentum of the black hole. There is an inner and an outer event horizon for a Kerr black hole which is given the zeros of Δ.

$$r_+ = M + (M^2 + a^2)^{1/2}$$
$$r_- = M - (M^2 + a^2)^{1/2}$$

The scalar curvature for the Kerr geometry is determined by contracting the Riemannian curvature tensor twice.

$$R = R^i_{\ i} = g^{il}R_{li}$$
$$R_{jl} = R^i_{\ jil}$$

where the Riemannian curvature tensor is given by

$$R^i_{\ jkl} = \Gamma^i_{\ jl,k} - \Gamma^i_{\ jk,l} + \Gamma^i_{\ mk}\,\Gamma^m_{\ jl} - \Gamma^i_{\ ml}\,\Gamma^m_{\ jk}$$

and the Christoffel symbols given by

$$\Gamma^i_{\ jk} = {}^1\!/_2\, g^{il}(g_{lj,k} + g_{lk,j} - g_{kj,l})$$

with the Kerr metric given by g_{ij} and its inverse by g^{ij}.

The contravariant components to the metric will be determined numerically by finding the matrix inverse of g_{ij}. The partial derivatives of the metric $g_{lj,k}$ and the Christoffel symbols $\Gamma^{i}_{jk,l}$ will also be determined numerically using finite differences. The angular momentum per mass is chosen to be a = 0.99 and the mass of the black hole is M = 1.00. The scalar curvature is calculated on a rectangular grid in the equatorial plane extending from -10M to 10M in both directions. Here is the GRPP source code for this example.

```
/*
  File:example4.g
  Author:TensorSoft
  Date:December 1993
  Purpose:Determine curvature for Kerr black hole

  Copyright (c) 1993-1999
  By TensorSoft
  All Rights Reserved
*/

$$ /* set up coordinates and indices used by GRPP */
coordinates$(t,r,a,p);
indices$(i,j,k,l,m);
$$

#include <stdio.h>
#include <math.h>

#define MASS  1.00/* Mass of rotating black hole */
#define ANGM  0.99/* Angular momentum per mass of rotating black hole */
#define EPS   1.0e-4/* Epsilon, a small number */
#define PI    3.14159265358979323846264433

double **array4x4; /* global 4x4 array for inverse metric calls */
void inverse(double**,int);
double **matrix(int,int,int,int);

void kerr_metric(radius, theta, phi, g_cc)
double radius, theta, phi;
/* last argument was Tensor */
$$ rank2 *g$ij ; $$
{
 double r2, A2, cos2a, sin2a, sigma, delta;
 double gtt, grr, gaa, gpp, gtp;

/* build up common terms in kerr metric */
 r2 = radius * radius; A2 = ANGM * ANGM;
 cos2a = cos( theta ); cos2a *= cos2a;
 sin2a = sin( theta ); sin2a *= sin2a;
 sigma = r2 + A2 * cos2a; delta = r2 - 2 * MASS * radius + A2;

/* do scalar algebra here, C is better at that! */
 gtt = - (1.0 - 2 * MASS * radius / sigma);
 grr = sigma / delta;
 gaa = sigma;
 gpp = (r2 + A2 + 2 * MASS * radius * A2 * sin2a / sigma) * sin2a;
 gtp = - 2 * ANGM * MASS * radius * sin2a / sigma;

$$ /* assign all components of kerr metric tensor */
 *g$tt = gtt;
 *g$rr = grr;
 *g$aa = gaa;
 *g$pp = gpp;
 *g$tp =+= gtp;
 *g$tr =+= 0.0;
 *g$ta =+= 0.0;
 *g$ra =+= 0.0;
```

```
 *g$rp =+= 0.0;
 *g$ap =+= 0.0;
$$
}

void kerr_christoffel(radius, theta, phi, Christoffel_Ccc)
double radius, theta, phi;
$$ rank3 *Christoffel$Ijk; $$
{
$$
 rank2 g$ij, g$IJ;
 rank2 g_r_plus_eps$ij, g_r_minus_eps$ij ;
 rank2 g_a_plus_eps$ij, g_a_minus_eps$ij ;
 rank3 g$ij,k;
$$
 void kerr_metric(), inverse();

 kerr_metric(radius, theta, phi, &g_cc);
$$ array4x4[] = g$ij; $$
 inverse(array4x4,4);
$$ g$IJ = array4x4[]; $$
 radius += EPS;
 kerr_metric(radius, theta, phi, &g_r_plus_eps_cc);
 radius -= 2*EPS;
 kerr_metric(radius, theta, phi, &g_r_minus_eps_cc);
 radius += EPS;
 theta  += EPS;
 kerr_metric(radius, theta, phi, &g_a_plus_eps_cc);
 theta  -= 2*EPS;
 kerr_metric(radius, theta, phi, &g_a_minus_eps_cc);
 theta  += EPS;
$$
 g$ij,t = 0.0;
 g$ij,r = ( g_r_plus_eps$ij - g_r_minus_eps$ij ) / (2 * EPS);
 g$ij,a = ( g_a_plus_eps$ij - g_a_minus_eps$ij ) / (2 * EPS);
 g$ij,p = 0.0;
 *Christoffel$Ijk = 0.5 * g$IL * ( g$lj,k + g$lk,j - g$kj,l );
$$
}

double kerr_curvature(radius, theta, phi)
double radius, theta, phi;
{
 double R;
$$
 rank2 g$ij, g$IJ, R$ij;
 rank3 Chr$Ijk;
 rank3 Chr_r_plus_eps$Ijk, Chr_r_minus_eps$Ijk;
 rank3 Chr_a_plus_eps$Ijk, Chr_a_minus_eps$Ijk;
 rank4 R$Ijkl, Chr$Ijk,l;
$$
 void kerr_metric(), kerr_christoffel(), inverse();

 kerr_metric(radius, theta, phi, &g_cc);
$$ array4x4[] = g$ij; $$
 inverse(array4x4,4);
$$ g$IJ = array4x4[]; $$
 kerr_christoffel(radius, theta, phi, &Chr_Ccc);
 radius += EPS;
 kerr_christoffel(radius, theta, phi, &Chr_r_plus_eps_Ccc);
 radius -= 2*EPS;
 kerr_christoffel(radius, theta, phi, &Chr_r_minus_eps_Ccc);
 radius += EPS;
 theta  += EPS;
 kerr_christoffel(radius, theta, phi, &Chr_a_plus_eps_Ccc);
 theta  -= 2*EPS;
 kerr_christoffel(radius, theta, phi, &Chr_a_minus_eps_Ccc);
 theta  += EPS;
$$
 Chr$Ijk,t = 0.0;
```

```
 Chr$Ijk,r = ( Chr_r_plus_eps$Ijk - Chr_r_minus_eps$Ijk ) / (2 * EPS);
 Chr$Ijk,a = ( Chr_a_plus_eps$Ijk - Chr_a_minus_eps$Ijk ) / (2 * EPS);
 Chr$Ijk,p = 0.0;
 R$Ijkl = Chr$Ijl,k - Chr$Ijk,l + Chr$Imk * Chr$Mjl - Chr$Iml * Chr$Mjk;
 R$jk = R$Ijik;
 R = g$IL * R$li;
$$
 return( R );
}

main()
{
 int x,y;
 double radius,theta,phi,R;
 double kerr_curvature();
 FILE *pfile;

 pfile = fopen("results.dat","w");
 theta = PI/2;
 array4x4 = matrix(0,3,0,3);
 for (x = -10; x <= 10; x++) {
 for (y = -10; y <= 10; y++) {
    radius = sqrt((double)(x*x) + (double)(y*y));
    phi = atan2((double)(y), (double)(x));
    if (radius > 0)
      {
       R = kerr_curvature(radius, theta, phi);
       fprintf(pfile,"%d %d %g \n",x,y,R);
      }
    else
      fprintf(pfile,"%d %d %g \n",0,0,0.0);
 } }
 free_matrix(array4x4,0,3,0,3);
 fclose(pfile);
}
```

A unix Makefile is used to build the executable. This facilitates building the object files for the general purpose routines, **inverse**, **matrix** and **free_matrix**. Here is the Makefile.

```
NAME    = example4
CFILES  = allocate.c inverse.c example4.c
CFLAGS  = -O
LDFLAGS =

LIBS    =

# Rules...

SRCFILES = $(CFILES)
OBJFILES = $(CFILES:.c=.o)

$(NAME): $(OBJFILES)
        $(CC) -o $@ $(CFLAGS) $(OBJFILES) $(LIBS) $(LDFLAGS)

example4.c: example4.g
          grpp -o example4.c example4.g
```

Building the target executable with the Makefile results in the following terminal session.

```
einstein> make
cc -O -c allocate.c
cc -O -c inverse.c
grpp -o example4.c example4.g

General Relativity PreProcessor
        Version 2.1

    Copyright (c) 1992-1999
    by TensorSoft
    All Rights Reserved

Results:
    example4.g: 166 total lines
    example4.c: 2421 total lines

cc -O -c example4.c
cc -o example4 -O allocate.o inverse.o example4.o
einstein> example4
einstein> ls
Makefile        example4.c      grcc*           results.dat
allocate.c      example4.g      grpp*
allocate.o      example4.o      inverse.c
example4*       example4_grpp.h  inverse.o
einstein>
```

Terminal

The calculated values for the scalar curvature are written to the file ‘results.dat’. They should be close to zero for this geometry. Inspection of the scalar curvature values shows it to lie randomly near zero, with the largest errors at the black hole. A very good result but it is sensitive to the value of EPS, as expected. A plot of the scalar curvature for the equatorial plane of the Kerr black hole is shown below.

Scalar curvature in equatorial plane of Kerr geometry